版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

Broadview®
www.broadview.com.cn

网银互联

一本由网银互联发起，集合了网易、美丽联合集团、酷家乐、个推、纵横新创、沃趣科技、搜道网、九言科技等多家公司容器技术实践案例的干货好书。

互联网企业
容器技术实践

龚曦◎主编

中国工信出版集团　电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

互联网企业
容器技术实践

龚曦◎主编

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书通过容器技术领域的实践者分享各自的实践案例，介绍常见的业务痛点、实现方式、方案的选型、遇到的问题和解决方案等，希望能够帮助读者了解、认识和玩转容器。本书主要分为两部分：第一部分是原理篇，包括第1章和第2章，简单介绍Docker和Kubernetes的基础知识及原理，包括Docker和Kubernetes是什么、可以做什么及如何使用等；第二部分是案例篇，包括第3～10章，通过多个实战案例，针对不同的使用场景和业务需求，介绍如何应用容器技术及实现相关需求。本书案例均来自一线真实案例，并有技术人员对技术改造过程的体会和领悟，有较强的借鉴意义和参考价值。

本书可供系统架构师、开发和测试人员、运维人员阅读，也可供企业IT主管、系统管理员、平台管理员参考。

图书在版编目（CIP）数据
互联网企业容器技术实践 / 龚曦主编. —北京：电子工业出版社，2019.1
ISBN 978-7-121-35004-7

Ⅰ. ①互… Ⅱ. ①龚… Ⅲ. ①互联网络 Ⅳ. ①TP393.4

中国版本图书馆CIP数据核字（2018）第207737号

策划编辑：陈晓猛　宋亚东
责任编辑：宋亚东
印　　刷：天津画中画印刷有限公司
装　　订：天津画中画印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路173信箱　　　邮编：100036
开　　本：787×1092　1/16　　印张：17　　字数：428.8千字
版　　次：2019年1月第1版
印　　次：2019年1月第1次印刷
定　　价：69.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　　言

随着互联网行业的不断发展，企业管理的服务器数量大幅增加，业务系统越来越复杂，用户体验要求也越来越高，运维工作面临的挑战和难度也越来越大。

2013年，Docker作为一个开源项目横空出世，解决了服务器应用快速构建、部署和分享的问题，能够把服务器应用像App一样简单地安装到各种平台环境中，而不受真实环境的影响。对于开发者和运维工程师来说，Docker是提高开发和部署效率的一大利器；而对于企业来说，能够节省大量的基础设施投入和降低维护成本。因此，Docker一出现就迅速引爆了技术圈，也再次将容器技术带回到人们的视野中。而Kubernetes作为后起之秀，在Docker技术的基础上，为容器化的实施提供了部署运行、资源调度、服务发现和动态伸缩等一系列完整功能，使得大规模容器集群管理更加便捷。虽然Kubernetes的出现比Docker晚一些，但其影响力同样巨大，作为一个容器编排管理平台，其受关注度甚至超过了容器本身。

容器技术的兴起，不仅使应用开发部署发生了深刻变革，也让应用设计架构和运维部署发生了新的变化。企业进行容器化改造，通常根据企业的实际应用和业务需求选择容器化管理平台，并进行应用容器化改造和DevOps建设。然而在此过程中，技术人员往往面临各种各样的问题，如平台的标准化、容器技术涉及的资源和应用内部结构、传统应用如何改造等。虽然这些问题有一些可借鉴的经验，但更多需要技术团队根据自身的情况去摸索解决。

如今，虽然市面上不乏关于容器的书籍，技术社区也一直很活跃，但真正应用容器技术的企业仍然不多，系统介绍具体落地案例的资料比较少，远不能满足企业和技术人员的需求。为此，我们特邀请了容器技术领域的实践者来撰写本书，分享他们的实践案例，介绍包括业务痛点、实现方式、方案选型、遇到的问题及解决方案等，希望能够将基于Docker和Kubernetes打造的企业级容器云平台的经验分享给广大读者。

本书主要内容

本书主要分为两部分：第一部分是原理篇，包括第1章和第2章，介绍Docker和Kubernetes的基础知识及原理，包括Docker和Kubernetes是什么、可以做什么及如何使用等；第二部分是案例篇，包括第3～10章，通过具体的实战案例，针对不同的使用场景和业务需求，介绍如何应用容器技术及实现相关需求。

本书特色

- **以实践和应用为导向。**本书抛开繁杂的理论讲解，从具体实践案例入手，由浅入深地介绍容器化的技术改造过程，并总结其中遇到的各种问题及相应的解决方案，涉及关键技术方案及创新点。同时，对容器云平台的功能组件有详细的技术架构设计，并对开源软件如何选型给出具体建议及应用场景分析，为容器云平台的具体实现提供方案。

- **清晰而有深度的讲解。**本书内容全部来自容器化改造一线的技术工程师之手，在编写过程中尽量用简单的语言表述，力求条理清晰，做到通俗易懂。本书除了对改造过程中的经验进行了总结，还对未来容器云生成环境的路线与规划进行了探讨。

致谢

本书由多位作者共同编写而成，包括（按姓氏拼音排序）：陈东辉（酷家乐运维部门负责人）、陈京来（网易云计算工程师）、付铖（酷家乐国际站技术负责人）、何晓波（网银互联云计算架构师）、李成（网银互联云计算工程师）、陆兵斌（酷家乐技术专家）、牛庆功（酷家乐研发工程师）、裘宏骏（纵横新创CTO）、苏茶林（九言科技运维部门负责人）、沃趣科技原型团队、吴叶磊（酷家乐研发工程师）、徐常雨（搜道网技术副总裁）、俞锋锋（个推应用平台研发部总监）、张振华（美丽联合集团虚拟化部门负责人）。正是他们牺牲了自己的休息时间，才能将项目实践方法总结成文字，一起促成了本书的合作和出版，以飨读者，在此向所有作者表示衷心的感谢！

同时也感谢电子工业出版社的陈晓猛和宋亚东编辑，得益于他们专业的工作和细致的帮助，让本书能够顺利出版发行。

虽然我们对书中的内容进行了仔细的编辑和校对，但难免会存在一些纰漏，欢迎广大读者批评指正，并与我们进行交流和探讨！

编者

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。
- 提交勘误：您对书中内容的修改意见可在“提交勘误”处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方“读者评论”处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35004

作 者 简 介

李成，网银互联资深云计算工程师。拥有多年云计算开发经验，长期关注云计算领域，熟悉虚拟化和容器等技术。

陈京来，来自网易伏羲AI实验室。专注云计算、人工智能领域的实践，长期关注虚拟化、容器化、深度学习等技术的发展，目前主要研究人工智能与云场景的结合落地。

张振华，花名郭嘉，美丽联合集团虚拟化部门负责人，10余年软件研发和技术管理经验，曾在英特尔、思科等公司工作。目前工作聚焦在美联容器云平台的研发和容器DevOps项目的落地上，在Docker容器、Kubernetes、DevOps、Cloud Native、微服务化等方面都有丰富的实战经验。

付铖，花名橙子，酷家乐国际化业务后端负责人，酷家乐早期员工，本科毕业于浙江大学软件工程系。酷家乐国际化业务现已全面应用K8S容器编排和Istio服务网格技术。

陆兵斌，花名斑斑，毕业于浙江大学软件工程专业，后就职于酷家乐。在酷家乐负责过云渲染系统、监控系统、发布系统等，在基础架构方面有丰富经验。现担任酷家乐全屋定制业务后端技术负责人。

牛庆功，花名东悦，毕业于中科大软件工程专业，毕业后一直从事容器云研发，目前负责酷家乐容器云研发，对Kubernetes、容器及应用有深入理解。

吴叶磊，花名阿磊，毕业于浙江大学数字媒体技术专业。就职于酷家乐，曾负责自研数据同步套件、配置中心等中间件的开发，目前负责监控警报系统，对Kubernetes、Prometheus有深入应用与理解。

俞锋锋，曾就职百度，先后负责百度手机助手PC端和手机百度新闻资讯等产品。现为个推应用平台研发部总监，具有丰富的技术管理和架构经验。

何晓波，杭州网银互联科技股份有限公司研发部负责人，拥有多年的软件研发和技术管理经验，在Kubernetes、云计算、虚拟化等方面均有丰富的实战经验。

徐常雨，搜道网技术副总裁，负责公司产品研发。长期从事互联网产品研发工作，拥有10年以上产品研发团队管理经验，对互联网各类型产品形态开发具有丰富的实战经验，长期关注高性能可扩展微服务架构设计和快速团队开发。目前从事营销自动化SaaS服务、大数据及人工智能领域相关产品的研发工作。

裘宏骏，纵横新创技术总监CTO，拥有15年以上的软件行业经验，从通信应用、互联网应用、微服务平台的架构设计和开发，到自动化构建、持续集成、持续交付及DevOps的转型实施都有较丰富的经验。

苏茶林，花名海天，九言科技运维部门负责人，长期从事系统保障工作，拥有10年运维经验，在系统、网络方面有着丰富的经验，曾就职于中国联通、杭州斯凯网络，目前就职于九言科技，负责推进公司容器化实施。

沃趣原型团队，沃趣科技是一家专注为企业用户提供基于高性能、高可用、可扩展的开放数据库云平台解决方案的国产厂商。公司创始团队为阿里巴巴原数据库技术团队核心骨干，凭借丰富的研发及运维经验，为行业客户提供数据库云产品及软硬件一体化解决方案。

目　录

CHAPTER

1

第 1 章
Docker 入门

本章主要围绕 Docker 容器来展开，首先介绍什么是容器，以及容器的发展历史和优势，并着重介绍 Docker 容器；然后介绍 Docker 容器实现的原理，以及一些 Docker 依赖的底层技术，如 Namespace、Cgroups 等；接着介绍什么是 Docker 镜像，如何构建 Docker 镜像，以及搭建 Docker 镜像仓库的方法等；最后介绍 Docker 容器的网络原理。

本章主要通过学习 Docker 容器帮助读者了解什么是容器，了解 Docker 容器的原理和一些基础操作，为学习 Kubernetes 打下基础。

1.1 容器简介与 Docker 容器引擎

1.1.1 什么是容器

容器又称为容器虚拟化技术，从名称可以看出容器也是一种虚拟化技术。容器是基于操作系统的轻量级的虚拟化技术。容器的英文名称是 Container，该单词的另一个意思是集装箱，其实容器和集装箱确实有很多相似的地方。下面就通过集装箱这个例子来讲一下什么是容器。如果把容器比作集装箱，容器上面运行的应用服务比作集装箱里面的货物，则运载货物的轮船就是物理机。集装箱可以将不同的货物隔离开来，只不过容器隔离的是应用服务罢了 。

其实容器就是将软件打包到一个平台中，用来开发、发布和部署的一种工具。容器镜像是一个轻量级、独立、可执行的软件包，包含运行它所需的所有内容：代码、运行时、系统工具、系统库、设置等。不管环境如何，容器化软件在 Linux 系统和 Windows 系统平台上都可以运行相同的应用程序。容器可以将软件与其周围环境隔离开来，解决开发环境和安装环境之间存在差异的问题，还有助于减少团队之间在同一基础架构上运行不同软件的冲突。

1.1.2 Docker 容器

一说到容器，几乎立马想到的就是 Docker 了，Docker 似乎已经成为容器的代名词，导致现在很多人以为容器就是 Docker。其实 Docker 只是容器的其中一种而已，它将容器技术发扬光大。

1. Docker 的由来

2013 年，dotCloud 公司决定开源他们内部使用的容器技术，这就是 Docker 的由来。之后，Docker 很快在各大论坛火爆起来。到了 2013 年 10 月，dotCloud 公司甚至直接更名为 Docker 股份有限公司，工作重心也转向全面围绕 Docker 来开发，可见 Docker 的火爆程度与崛起速度。Docker 之所以能迅速崛起，是因为它解决了传统方式的各种问题，在很大程度上提高了软件的生产效率，这是很多用户在生产环境中真实遇到的痛点。

2. Docker 是什么

其实 Docker 并不能算一种新技术，因为 Docker 是基于 LXC（Linux Containers），并使用了一些已经存在了很长的时间并且被广泛应用的技术来实现的。Docker 主要是将这些复杂、琐碎的底层技术服务化了。之所以能够获得如此巨大的成功，受到如此多用户的青睐，因为它能够更加了解用户的需求，且足够的简单方便。下面演示 Docker 的使用，运行 Docker 的最小镜像 hello-world。如果使用的系统环境已经安装了 Docker，则可以直接在 Root 环境下执行以下命令：

```
docker run hello-world
```

其实这个镜像并没有实际意义，只是 Docker 的最小镜像，最大的作用是用来测试 Docker 服务环境是否正常，在一切正常的情况下，首行会打印出“Hello from Docker!”的字符串。

3. Docker 的一般操作

本节主要讲解 Docker 的一些常用操作，读者可以跟着动手操作。

既然要进行操作，首先需要具备 Docker 环境，下面介绍 Docker 的安装方法。

（1）安装 Docker

Docker 安装起来很方便，其主要是安装在 Linux 系统环境中，因为 Docker 的实现是依赖 Linux 内核的。当然，Docker 也有 Windows 和 Mac OS 版本，但实质是安装了一个 Linux 虚拟机，所以为了获得最好的体验，还是需要 Linux 环境的。本书在 CentOS 环境中演示：

```
# yum upgrade
# yum install docker
# systemctl enable docker
# systemctl start docker
```

（2）拉取镜像

Docker 通过 docker pull 命令拉取镜像，命令格式如下：

```
docker pull [选项] [Docker Registry 地址[:端口号]/]仓库名[:标签]
```

如上所示，Docker 仓库的默认地址是 Docker Hub，当没有明确指出时，就会从 Docker Hub 中拉取镜像。同时，仓库名是两段式名称，采用<用户名>/<镜像名>这种格式，当没有给出用户名的时候，默认为 library，即为官方镜像。下面拉取一个 ubuntu:16.04 镜像。

```
# docker pull ubuntu:16.04
16.04: Pulling from library/ubuntu
297061f60c36: Pull complete
e9ccef17b516: Pull complete
dbc33716854d: Pull complete
8fe36b178d25: Pull complete
686596545a94: Pull complete
Digest: sha256:1dfb94f13f5c181756b2ed7f174825029aca902c78d0490590b1aaa203abc052
Status: Downloaded newer image for ubuntu:16.04
```

在没有指明 Docker 镜像仓库地址的情况下，会将标签为 16.04 的镜像从 Docker Hub 中的官方镜像仓库 library/ubuntu 中拉取到本地。有了 Docker 镜像，就可以通过镜像运行容器。

（3）镜像的保存和加载

有时会由于环境的原因，导致有些机器无法拉取镜像，可以使用 Docker 镜像的保存加载机制，把镜像复制到任意的服务器上。

首先通过 docker save 命令将一个本地存在的镜像打包成一个 tar 文件，如下所示。

```
# docker save  ubuntu:16.04 > ubuntu-1604.tar

# ls
ubuntu-1604.tar
```

从上面的代码可以看到，已经将本地的 ubuntu:16.04 镜像打包成 ubuntu-1604.tar 并保存在本地，接下来可以把 ubuntu-1604.tar 打包文件复制到另外一台服务器上，并通过 docker load 将 ubuntu:16.04 镜像加载到这台服务器上，如下所示。

```
# docker load < ubuntu-1604.tar
Loaded image: ubuntu:16.04

# docker images
REPOSITORY     TAG             IMAGE ID              CREATED              SIZE
ubuntu          16.04          0b1edfbffd27          4 weeks ago          113MB
```

（4）容器的导入和导出

容器的导入和导出与镜像的保存和加载有很多相似之处，不同之处在于一个用于操作容器，另一个用于操作镜像。

首先通过 docker export 命令将一个本地存在的容器打包成一个 tar 文件，如下所示。


```
# docker ps -a
CONTAINER ID  IMAGE          COMMAND       CREATED         STATUS      PORTS         NAMES
d6dfb1953664  ubuntu:16.04  "/bin/bash"   5 seconds ago  Exited (0) 3 seconds ago   tender_fermat

# docker export tender_fermat > ubuntu-export-1604.tar

# ls
ubuntu-export-1604.tar
```

如上所示，已将本地的容器 tender_fermat 打包成 ubuntu-export-1604.tar 文件，接着可以把该文件复制到其他的服务器上，然后在服务器上通过 import 命令将容器快照导入为镜像，如下所示。

```
# docker import ubuntu-export-1604.tar
sha256:0abe0dfe67e991d7af8fbaaed6f03eef4471cad4d8358f9ae7421a957867dc2a

# docker image ls
REPOSITORY      TAG                 IMAGE ID            CREATED             SIZE
<none>          <none>              0abe0dfe67e9        40 seconds ago      85.9MB
```

（5）启动 Docker 容器

启动 Docker 容器有两种类型，一种是通过镜像创建并启动一个新的容器，另一种是启动一个处于停止状态的容器。可以通过 docker run 命令创建并启动一个容器，如下所示。

```
# docker run nginx
```

如上启动了一个 Nginx 服务器的容器，下面重新打开一个终端界面，通过 docker ps 命令查看本地启动的容器：

```
# docker ps
CONTAINER ID   IMAGE   COMMAND    CREATED       STATUS                          PORTS   NAMES
29ce0a9cbc17   nginx   "nginx -g  'daemon of…" 17 seconds ago Up 16 seconds  80/tcp  determined_
kowalevski
```

由上面可以看出本地启动了一个 Nginx 服务器容器。要启动一个处于停止状态的容器，可以使用 docker start 命令。先用 Ctrl+C 组合键终止刚启动的容器，如下所示。

```
# docker run nginx
^C#
```

可以通过 docker ps -a 命令查看本地所有的容器：

```
# docker ps -a
CONTAINER ID  IMAGE  COMMAND    CREATED       STATUS       PORTS                      NAMES
29ce0a9cbc17  nginx  "nginx -g 'daemon of…"  10 minutes ago  Exited (0) 2 minutes ago  determined_
kowalevski
```

可以发现容器已经终止，下面用 docker start 命令启动该容器，如下所示。

```
# docker start determined_kowalevski
determined_kowalevski

# docker ps
CONTAINER ID  IMAGE   COMMAND                  CREATED          STATUS        PORTS   NAMES
29ce0a9cbc17  nginx   "nginx -g 'daemon of…"  13 minutes ago  Up 4 seconds  80/tcp  determined_
kowalevski
```

需要注意的是，在大多数情况下启动的容器需要使其在后台运行，可以通过当 docker run 启动容器时，为其添加一个-d 参数来实现。这样容器启动后就不会占用一个终端，生命周期不会受到该终端生命周期的影响，同时也就无法通过 Ctrl+C 组合键终止容器进程了。

（6）进入容器

因为大多数情况都是通过-d 参数使容器在后台启动的，所以需要进入容器中才能进行一些操作。Docker 进入容器可以通过 docker attach 命令或 docker exec 命令两种方式，通常推

荐使用 docker exec 的方式。下面通过 docker attach 的方式进入，如下所示。

```
# docker attach determined_kowalevski
^C#

# docker ps -a
CONTAINER ID  IMAGE  COMMAND                  CREATED         STATUS          PORTS    NAMES
29ce0a9cbc17  nginx  "nginx -g 'daemon of…" 28 minutes ago  Exited (0) 29 seconds ago   determined_
kowalevski
```

可以看出，通过 docker attach 进入容器后退出时，会导致容器终止。因为当使用 docker attach 命令进入容器时，执行的是容器自带的命令，把容器默认进程杀掉，容器自然就退出了。下面使用 docker exec 方式进入容器：

```
# docker exec -it determined_kowalevski /bin/bash
root@29ce0a9cbc17:/# exit
exit

# docker ps
CONTAINER ID  IMAGE  COMMAND                  CREATED         STATUS          PORTS    NAMES
29ce0a9cbc17  nginx  "nginx -g 'daemon of…" 34 minutes ago  Up About a minute  80/tcp  determined_
kowalevski
```

通过 docker exec 进入容器并退出，对容器不会产生影响。在如上命令中，其中的-i 参数指打开容器的标准输入，-t:告诉 Docker 为容器建立一个命令行终端，而/bin/bash 则表示进入容器中要执行的命令。

（7）终止容器

Docker 可以使用 docker stop 命令终止一个正在运行的容器，如下所示。

```
# docker ps
CONTAINER ID  IMAGE  COMMAND                  CREATED         STATUS          PORTS    NAMES
29ce0a9cbc17  nginx  "nginx -g 'daemon of…"  34 minutes ago  Up About a minute  80/tcp  determined_
kowalevski

# docker stop determined_kowalevski
determined_kowalevski

# docker ps -a
CONTAINER ID  IMAGE  COMMAND                  CREATED         STATUS          PORTS    NAMES
29ce0a9cbc17  nginx  "nginx -g 'daemon of…" 40 minutes ago  Exited (0) 4 seconds ago   determined_
kowalevski
```

（8）删除容器

Docker 可以通过 docker rm 命令删除一个已经终止的容器，如下所示。

```
# docker rm determined_kowalevski
determined_kowalevski

# docker ps -a
CONTAINER ID    IMAGE      COMMAND        CREATED       STATUS        PORTS          NAMES
```

1.2 Docker 核心原理

1.2.1 Docker 引擎结构

Docker 引擎的结构如图 1-1 所示，是一组包含三大组件的客户端-服务器应用程序。

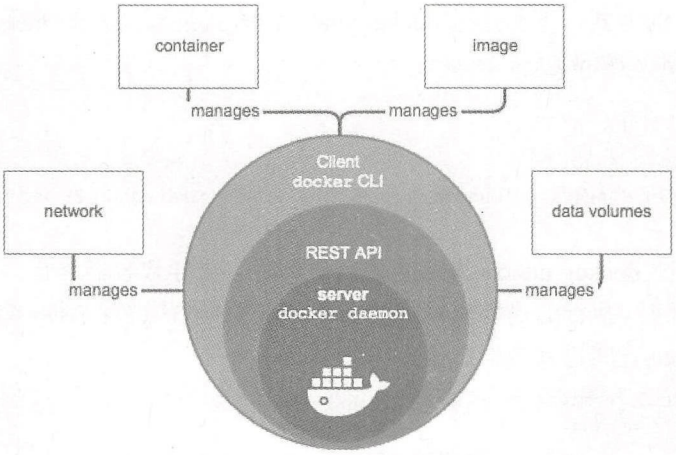


图 1-1　Docker 引擎的结构

Docker daemon（server）是一种称为守护进程（dockerd 命令）的长期运行程序；REST API 用于指定程序用来与守护进程对话并指示它执行操作的接口；Docker CLI（Client）是命令行界面（CLI）客户端（docker 命令）。

1.2.2　Docker 服务流程

Docker 服务运行的流程如图 1-2 所示。Docker 客户端与 Docker 守护进程通信，Docker 守护进程负责构建、运行和分发 Docker 容器。Docker 客户端和守护进程可以在同一个系统上运行，也可以将 Docker 客户端连接到远程 Docker 守护进程。Docker 客户端和守护进程使用 REST API 通过 UNIX 套接字或网络接口进行通信。

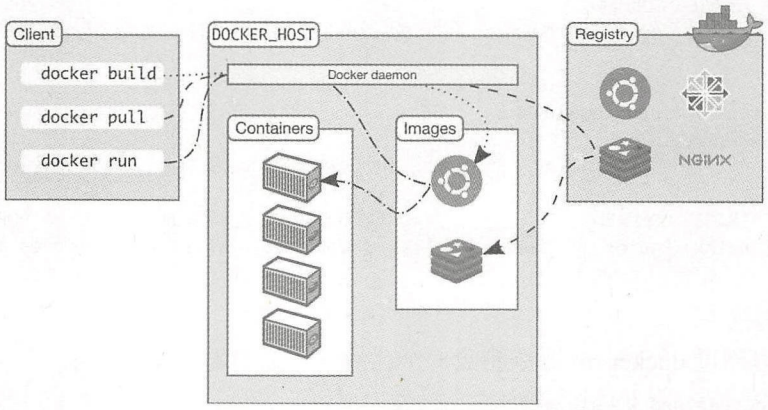


图 1-2　Docker 服务运行的流程

1.2.3　Docker 核心技术

前文提到 Docker 主要是使用了一些已有的技术实现的，主要的核心技术是 Cgroup+Namespaces 和 UnionFS。

1. Namespaces

命名空间（Namespaces）是 Linux 系统提供的一种内核级别的环境隔离方法，Docker 就是利用的这种技术来实现容器环境隔离的。Linux Namespace 提供了对 UTS、IPC、mount、PID、network、User 等的隔离机制。

（1）UTS Namespace

UTS Namespace主要是用来隔离hostname和domainname两个系统标识的，可以通过Go语言创建一个UTS Namespace，如下所示。

```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    cmd := exec.Command("bash")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
}
```

上述代码需要在Root权限下执行。在编译并执行后进入新的UTS Namespace。

```
# echo $$
20011
$ pstree -pl | grep 20011
    |  |  |  |  |-bash(16676)---sudo(200, 05)---mydocker(20006)-+-bash(20011)
```

当前PID为20011，再通过pstree查看其进程树，可知其父进程是20006。下面验证是否在同一个UTS Namespace下。

```
# readlink /proc/20011/ns/uts
uts:[4026532628]
# readlink /proc/20006/ns/uts
uts:[4026531838]
# readlink /proc/1/ns/uts
uts:[4026531838]
```

可以看到当前进程和父进程不在一个UTS Namespace下，父进程和1号进程在同一个UTS Namespace下，说明当前UTS Namespace是新创建的。下面测试UTS Namespace对hostname的隔离功能。

```
# hostname -b docker
# hostname
docker
# exit
exit
$  hostname
licheng-thinkpad-s2
```

可以看到，在新的UTS Namespace下先修改hostname为docker，查看修改成功，退出当前Namespace，或者打开一个新的终端，查看hostname还是原来的hostname，外部的hostname并没有受到影响。由此可以看到UTS Namespace的隔离作用。

（2）IPC Namespace

IPC Namespace主要提供了进程间通信的隔离能力。同样可以用Go语言实现IPC Namespace的创建。代码与创建UTS Namespace的基本相同，只要把标识符CLONE_NEWUTS换成

CLONE_NEWIPC 即可。

同样，编译后在 Root 环境下启动后就可以进入新的 IPC Namespace。在此 Namespace 下，可以查看 IPC Namespace 与进程 1 的 IPC Namespace 不同，说明是新创建的 IPC Namespace。

```
# readlink /proc/$$/ns/ipc
ipc:[4026532628]
# readlink /proc/1/ns/ipc
ipc:[4026531839]
```

此时，在此被隔离的 ipc namespace 中创建一个消息队列。

```
# ipcmk --queue
消息队列 id: 0
# ipcs -q

--------- 消息队列 -----------
键         msqid      拥有者     权限       已用字节数   消息
0x9ea09b5b 0          root       644        0            0
```

在新建的 IPC Namespace 中已经有了一条消息队列 0x9ea09b5b。另外再起一个终端，确认是否能够看到刚创建的消息队列。

```
# ipcs -q

--------- 消息队列 -----------
键         msqid      拥有者  权限     已用字节数 消息
```

确实无法看到刚创建的消息队列，说明消息队列确实被 IPC Namespace 隔离起来了。

（3）PID Namespace

PID Namespace 用来隔离进程。同样的进程在不同的 PID Namespace 下拥有不同的 PID，通过代码创建一个新的 PID Namespace，同样只需要把 UTS Namespace 的代码做少量修改，把标识符修改为 CLONE_NEWPID 即可。

依旧编译后在 Root 下启动，通过命令查看在新的 PID Namespace 下此进程的 PID，如下所示。

```
# echo $$
1
```

如果使用 ps 命令查看，看到的还是所有的进程，因为/proc 文件系统（procfs）没有挂载到与原/proc 不同的位置。如果只想看到 PID Namespace 本身应该看到的进程，则需要重新挂载/proc，如下所示。

```
# ps
  PID TTY          TIME CMD
 6239 pts/2    00:00:00 sudo
 6240 pts/2    00:00:00 mydocker
 6245 pts/2    00:00:00 bash
 6259 pts/2    00:00:00 ps
# mount -t proc proc /proc
# ps a
  PID TTY          TIME CMD
    1 pts/19   00:00:00 bash
   12 pts/19   00:00:00 ps
```

可以看到在 PID Namespace 中只有 bash 和 ps 进程。进程通过 Namespace 隔离，需要注意的是，由于没有进行 Mount Namespace 的隔离，当退出当前 Namespace 再执行 ps 命令时，系统会报错，需要将/proc 目录重新“mount”回去。

（4）Network Namespace

Network Namespace 在 Docker 中被用来隔离网络。Network Namespace 可以让每个容器拥

有自己的网络设备、端口、IP 地址等。因为其网络是完全隔离的，所以每个 Namespace 下的端口不会产生任何冲突。既然完全隔离了，容器与外部之间需要通信该怎么办？在 Docker 中可以通过虚拟网桥来实现。在 Linux 系统中，可以直接通过命令创建一个 Network Namespace。当然，为了与上文保持一致，仍然通过代码来实现。代码实现方式与上文几个 Namespace 一样，只是把标识符改为 CLONE_NEWNET，然后将代码编译并执行，通过 ifconfig 命令查看其 Namespace 下的网络设备，如下所示。

```
# ifconfig
#
```

可以看到里面并没有任何的网络设备。然后在宿主机中查看宿主机的网络设备，可见 Network Namespace 网络隔离成功。

```
# ifconfig
docker0   Link encap:以太网  硬件地址 02:42:fb:e0:96:cf
          inet 地址:172.17.0.1  广播:172.17.255.255  掩码:255.255.0.0
          inet6 地址: fe80::42:fbff:fee0:96cf/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:7 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:1401 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:1175 (1.1 KB)  发送字节:268314 (268.3 KB)

eth0      Link encap:以太网  硬件地址 68:07:15:ef:30:3f
          inet 地址:192.168.2.199  广播:192.168.2.255  掩码:255.255.255.0
          inet6 地址: fe80::ed9:6938:55cc:1571/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:200495 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:109379 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:57381467 (57.3 MB)  发送字节:25676057 (25.6 MB)

lo        Link encap:本地环回
          inet 地址:127.0.0.1  掩码:255.0.0.0
          inet6 地址: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  跃点数:1
          接收数据包:8862 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:8862 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:1842580 (1.8 MB)  发送字节:1842580 (1.8 MB)
```

（5）Mount Namespace

Mount Namespace 用来隔离各个进程看到的挂载点视图。在不同 Namespace 中的进程看到的文件系统层次是不一样的，不同的 Namespace 进行 mount 和 umount 操作只会对自己的 Namespace 内的文件系统产生影响，对其他的 Namespace 没有影响。

由于没有进行 Mount Namespace，所以退出 PID Namespace 时才会报错。所以在 PID Namespace 的基础上再加上 Mount Namespace 进行测试。这个可以直接在 PID Namespace 的代码基础上再加上 Mount Namespace 的标识符。因为 Mount Namespace 是 Linux 实现的第一个 Namesapce 类型，其标识符比较特殊，是 CLONE_NEWNS，代码如下：

```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
```

```
)

func main() {
    cmd := exec.Command("bash")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWPID | syscall.CLONE_NEWNS,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
}
```

测试方式是在 Root 下编译并执行的，在此 Namespace 下执行挂载/proc 并执行 ps 命令，如下所示。

```
# mount --make-private /proc
# mount -t proc proc /proc
# ps
  PID TTY          TIME CMD
    1 pts/2    00:00:00 bash
   20 pts/2    00:00:00 ps
```

另起一终端，执行如下 ps 命令，可见两个不同的 Namespace 下的/proc 并不一样，说明 mount 已经隔离成功，在新的 Namespace 下的 mount 操作并没有影响到外部的 Namespace 下的系统文件。

```
# ps
  PID TTY          TIME CMD
14319 pts/20   00:00:00 bash
15103 pts/20   00:00:00 ps
```

（6）User Namespace

User Namespace 可以用来隔离用户的用户组 ID。在不同的 User Namespace 下进程的 User ID 和 Group ID 是不同的。同样通过 Go 语言来实现创建一个新的 User Namespace，代码与上面的基本相同，修改一个标识符为 CLONE_NEWUSER 即可。

首先在 Root 环境下查看宿主机当前的用户和用户组：

```
# id
uid=0(root) gid=0(root) 组=0(root)
```

可以看到是 root 用户。运行一下程序，创建新的 User Namespace，在新的 Namespace 下查看用户和用户组。

```
$ id
uid=65534(nobody) gid=65534(nogroup) 组=65534(nogroup)
```

2. Cgroups

（1）什么是 Cgroups

Cgroups 是 Linux 系统中提供的对一组进程及其子进程进行资源（CPU、内存、存储和网络等）限制、控制和统计的能力。Cgroup 可以直接通过操作 Cgroup 文件系统的方式完成使用。例如，使用 mount -t cgroup cgroup /cgroup 命令进行操作，此时就会在/cgroup 下生成很多默认的文件，这就创建一个 Cgroup，在这个目录下每创建一个目录就表示创建了一个子 Cgroup。进入子目录会发现里面会生成一些文件与上层 Cgroup 即/cgroup 目录内容大致相同。这就是 Cgroup 文件系统的树形层次结构。

创建完 Cgroup 之后，可以为其分配可用的资源并将不同的进程放进去。当创建完第一个 Cgroup 时，系统会把所有的进程都放到主 Cgroup 中，可以查看 Cgroup 中的 tasks 文件来查看此 Cgroup 中的进程 PID；同样可以通过在 tasks 中添加对应的进程 PID，会把该进程放入该 Cgroup 中。但需要注意，如果在子 Cgroup 中添加一个进程，则子 Cgroup 的上层 Cgroup 中的 tasks 文件中也会有这个 PID，因为子 Cgroup 属于上层 Cgroup，所以子 Cgroup 中的进程也同时会属于上层 Cgroup，但是同一层级的 Cgroup 却不能同时拥有同一个进程。比如 A 和 BCgroup 同属于 C 的子 Cgroup，那么 A 和 B 就不能同时拥有同一个进程。至于每个 Cgroup 中的资源配置量都是通过设置当前 Cgroup 的子系统来配置的。

Cgroups 为不同的资源定义了各自的 Cgroup 子系统，来实现对资源的具体管理。Cgroup 实现的子系统及其实现的功能如下。

- devices：设备权限控制。
- cpuset：分配指定的 CPU 和内存节点。
- cpu：控制 CPU 占用率。
- cpuacct：统计 CPU 的使用情况。
- memory：限制内存的使用上限。
- freezer：暂停 Cgroup 中的进程。
- net_cls：配合 traffic controller 限制网络带宽。
- net_prio：可以动态控制每个网卡流量的优先级。
- blkio：限制进程的块设备 I/O。
- ns：使不同 Cgroups 中的进程使用不同的 Namespace。

（2）Cgroups 的使用

前文提到可以使用 mount -t cgroup cgroup /cgroup 命令创建 cgroups，其中可以通过-o 参数添加子系统，如命令 mount -t cgroup -o cpu、cpuset、memory cgroup /cgroup。这个命令表示创建了一个名为 Cgroup 的层级，并附加了 cpu、cpuset、memory 三个子系统，并把层级挂载到 /cgroup 目录上。但实际执行命令时会报出 already mounted 错误，且执行不成功。这是因为该命令一般在 Linux 发行版启动时就已经执行了，对应的子系统的 Cgroup 已经被创建并挂载了。并且虽然 cgroupfs 可以挂载在任意目录中，但是标准挂载点是/sys/fs/cgroup 目录并且在启动时已经挂载上了，所以一般并不需要执行该命令。由于系统的/sys/fs/cgroup 目录已经挂载了各种 cgroupfs，可以直接在该 Cgroup 上进行操作。

首先查看/sys/fs/cgroup，如下所示。

```
/sys/fs/cgroup# ll
总用量 0
drwxr-xr-x 14 root root 360 5月  18 09:54 ./
drwxr-xr-x  9 root root   0 5月  18 10:55 ../
dr-xr-xr-x  5 root root   0 5月  18 10:55 blkio/
lrwxrwxrwx  1 root root  11 5月  18 09:54 cpu -> cpu,cpuacct/
lrwxrwxrwx  1 root root  11 5月  18 09:54 cpuacct -> cpu,cpuacct/
dr-xr-xr-x  5 root root   0 5月  18 10:55 cpu,cpuacct/
dr-xr-xr-x  2 root root   0 5月  18 10:55 cpuset/
dr-xr-xr-x  5 root root   0 5月  18 10:55 devices/
dr-xr-xr-x  2 root root   0 5月  18 10:55 freezer/
dr-xr-xr-x  2 root root   0 5月  18 10:55 hugetlb/
dr-xr-xr-x  5 root root   0 5月  18 10:55 memory/
lrwxrwxrwx  1 root root  16 5月  18 09:54 net_cls -> net_cls,net_prio/
dr-xr-xr-x  2 root root   0 5月  18 10:55 net_cls,net_prio/
lrwxrwxrwx  1 root root  16 5月  18 09:54 net_prio -> net_cls,net_prio/
```

```
dr-xr-xr-x  2 root root   0 5月  18 10:55 perf_event/
dr-xr-xr-x  5 root root   0 5月  18 10:55 pids/
dr-xr-xr-x  2 root root   0 5月  18 10:55 rdma/
dr-xr-xr-x  5 root root   0 5月  18 10:55 systemd/
```

可以看到/sys/fs/cgroup 目录下有很多子目录，分别对应拥有对应子系统的 Cgroup，以 cpuset 为例，查看 cpuset 目录，如下所示。

```
/sys/fs/cgroup# ls cpuset/
cgroup.clone_children  cpuset.memory_pressure
cgroup.procs           cpuset.memory_pressure_enabled
cgroup.sane_behavior   cpuset.memory_spread_page
cpuset.cpu_exclusive   cpuset.memory_spread_slab
cpuset.cpus            cpuset.mems
cpuset.effective_cpus  cpuset.sched_load_balance
cpuset.effective_mems  cpuset.sched_relax_domain_level
cpuset.mem_exclusive   notify_on_release
cpuset.mem_hardwall    release_agent
cpuset.memory_migrate  tasks
```

可以看到里面有很多的控制文件，其中以 cpuset 开头的是 cpuset 子系统产生的，剩下的是由 Cgroup 产生的。前文已经提到过默认所有进程的 PID 都是在 Cgroup 的根目录的 tasks 文件中，通过 mkdir 创建一个 childA 目录，就创建了一个子 Cgroup，如下所示。

```
/sys/fs/cgroup/cpuset# mkdir childA
```

接着进入 childA 目录对该子 Cgroup 进行配置，可以通过修改文件的方式进行配置，如下所示。

```
/sys/fs/cgroup/cpuset/childA# echo 0 > cpuset.cpus
/sys/fs/cgroup/cpuset/childA#  echo 0 > cpuset.mems
```

两个命令分别表示限制 Cgroup 里的进程只能在 0 号 CPU 上运行，并只会从 0 号内存节点分配内存。接下来是给 Cgroup 分配进程，上文也已经提到通过修改 tasks 的方式把进程添加到当前 Cgroup 中，如下所示。

```
/sys/fs/cgroup/cpuset/childA# echo $$ > tasks
```

上面的命令表示把当前进程添加到 Cgroup 中，其中$$变量表示当前进程的 PID。这时进程的所有子进程也会被自动地添加到 Cgroup 中，并受到该 Cgoup 资源的限制。

3. UnionFS

（1）什么是 UnionFS

UnionFS（联合文件系统）是把不同物理位置的目录合并到同一个目录中的文件系统服务。其早期是应用在 LiveCD 领域的，通过联合文件系统可以非常快速地引导系统初始化或检测磁盘等硬件资源。这是因为只需要把 CD 只读挂载到特定目录，然后在其上附加一层可读写的文件层，对文件的任何变动修改都会被添加到新的文件层内，这种技术被称为写时复制。

写时复制是一种可以有效节约资源的技术，它被很好地应用在 Docker 镜像上。其思想是如果有一个资源需要被重复利用，在没有任何修改的情况下，新旧实例会共享资源，并不需要进行复制，如果有实例需要对资源进行任何的修改，并不会直接修改资源，而是会创建一个新的资源并在其上进行修改，这样原来的资源并不会进行任何修改，而是与新创建的资源结合在外，表现为修改后的资源状态。这样做可以显著地减轻对未修改资源的复制而带来的资源消耗问题。下面通过讲解在 Docker 中如何使用 UnionFS 更深入地理解写时复制。

Docker 支持的第一种 UnionFS 是 AUFS，下面主要从镜像层和容器层两个方面介绍 AUFS

在 Docker 中的使用。

（2）AUFS 在 Images 中的使用

Docker 镜像是由一层层的只读层组合而成的。镜像层的内容存储在/var/lib/docker/aufs/diff 目录下，在/var/lib/docker/aufs/layers 目录下则存放着对应的 metadata，描述镜像需要的层。

```
sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
a48c500ed24e: Pull complete
1e1de00ff7e1: Pull complete
0330ca45a200: Pull complete
471db38bcfbf: Pull complete
0b4aba487617: Pull complete
Digest: sha256:c8c275751219dadad8fa56b3ac41ca6cb22219ff117ca98fe82b42f24e1ba64e
Status: Downloaded newer image for ubuntu:latest
```

拉取一个 ubuntu:latest 镜像，看到 ubuntu:latest 镜像是分为 5 层的，可以在/var/lib/docker/aufs/diff 目录中确认。

```
# ls
47c4ed02b668dcf61970686e8b42d9b9e8c5b76d0e75acb0cbd5ad15dfc6b9f2
824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976
ed7b3b5ba6c11b0bd8182b9c940fc5d2402d4cdfbedb946a411366d091da0199
6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed
e85c04785e023817c8e6b64c2784ae931e7d7c7e84e0be9f5559a70a64db83bf
```

可以看到在本地没有别的镜像的情况下，目录中确实有 5 层，而这 5 层是如何组合成镜像的呢？来看/var/lib/docker/aufs/layers 中的文件：

```
# ls
47c4ed02b668dcf61970686e8b42d9b9e8c5b76d0e75acb0cbd5ad15dfc6b9f2
824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976
ed7b3b5ba6c11b0bd8182b9c940fc5d2402d4cdfbedb946a411366d091da0199
6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed
e85c04785e023817c8e6b64c2784ae931e7d7c7e84e0be9f5559a70a64db83bf

# cat 47c4ed02b668dcf61970686e8b42d9b9e8c5b76d0e75acb0cbd5ad15dfc6b9f2
ed7b3b5ba6c11b0bd8182b9c940fc5d2402d4cdfbedb946a411366d091da0199
6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed
824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976

# cat 824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976

# cat ed7b3b5ba6c11b0bd8182b9c940fc5d2402d4cdfbedb946a411366d091da0199
6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed
824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976

# cat 6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed
824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976

# cat e85c04785e023817c8e6b64c2784ae931e7d7c7e84e0be9f5559a70a64db83bf
47c4ed02b668dcf61970686e8b42d9b9e8c5b76d0e75acb0cbd5ad15dfc6b9f2
ed7b3b5ba6c11b0bd8182b9c940fc5d2402d4cdfbedb946a411366d091da0199
6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed
824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976
```

由于层 ID 太长，截取前 5 个字符表示一下，可以看出 e85c0 文件中包含了剩下的所有的层，说明其是在文件的最上层依赖于下面的所有层；而 824dc 文件是空的所以是最基础、最底层的，所以整个镜像文件从最高层到最底层依次是 e85c0→47c4e→ed7b3→66915→824dc，再分别查看各目录的文件，如下所示。

```
/e85c04785e023817c8e6b64c2784ae931e7d7c7e84e0be9f5559a70a64db83bf# ls
run
```

```
    /47c4ed02b668dcf61970686e8b42d9b9e8c5b76d0e75acb0cbd5ad15dfc6b9f2# ls
    etc

    ed7b3b5ba6c11b0bd8182b9c940fc5d2402d4cdfbedb946a411366d091da0199# ls
    var

    6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed# ls
    etc  sbin  usr  var

    824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976# ls
    bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp
usr  var
```

根据上面各层的文件目录，可以推断出，在 build ubuntu:latest 的镜像过程中，最近的一次文件的修改是 run 目录下的，按时间顺序镜像推算，修改的文件所属目录为 bin,boot 等->etc,sbin,usr,var->var->etc->run。具体是不是这样呢？可以验证一下，如下是 ubuntu 镜像的 Dockerfile。

```
    FROM scratch
    ADD ubuntu-artful-core-cloudimg-amd64-root.tar.gz
    RUN set -xe \
        \
        && echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
        && echo 'exit 101' >> /usr/sbin/policy-rc.d \
        && chmod +x /usr/sbin/policy-rc.d \
        \
        && dpkg-divert --local --rename --add /sbin/initctl \
        && cp -a /usr/sbin/policy-rc.d /sbin/initctl \
        && sed -i 's/^exit.*/exit 0/' /sbin/initctl \
        \
        && echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \
        \
        && echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb
/var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' >
/etc/apt/apt.conf.d/docker-clean \
        && echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb
/var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' >>
/etc/apt/apt.conf.d/docker-clean \
        && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache "";' >>
/etc/apt/apt.conf.d/docker-clean \
        \
        && echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/docker-no-languages \
        \
        && echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTypes::Order:: "gz";' >
/etc/apt/apt.conf.d/docker-gzip-indexes \
        \
        && echo 'Apt::AutoRemove::SuggestsImportant "false";' >
/etc/apt/apt.conf.d/docker-autoremove-suggests
    RUN rm -rf /var/lib/apt/lists/*
    RUN sed -i 's/^#\s*\(deb.*universe\)$/\1/g' /etc/apt/sources.list
    RUN mkdir -p /run/systemd && echo 'docker' > /run/systemd/container
    CMD ["/bin/bash"]
```

从 Dockerfile 可以看到在制作 ubuntu 镜像的过程中共执行了 5 次文件修改命令，也可以看到该镜像是由 5 层组成的，每一层正好对应一次的文件修改。按照顺序文件，修改得越早，产生越早越在底层。由于在 ADD 添加一个压缩文件时会自动解压为目录，所以最开始修改的文件是 bin、boot 等，对应的是第一个 ADD 中添加的压缩文件 ubuntu-artful-core-cloudimg-amd64-root.tar.gz 解压出来的目录。再看第二条命令中一共修改了/usr、/sbin、/etc、/var 这些目录文件，正好和第二层修改的文件目录相同，剩下的每一层的目录也正好和 Dockerfile 中修改的文件目录一一对应。由此可以确定，在 Build 镜像中对文件的每一次修改都会增加一个文件层包含着

对应修改后的文件。而当进行删除文件操作时，也会创建一个新的层，在新层里创建一个特殊名称隐藏文件，这样的一个隐藏文件对应着一个文件的删除。

下面在 ubuntu:latest 镜像的基础上创建一个新的镜像 test，首先创建一个 test 文件。然后编辑 Dockerfile，生成镜像。

```
# echo "hello world" > test

# cat Dockerfile
FROM ubuntu:latest
COPY test /

# docker build -t test .
Sending build context to Docker daemon  2.008MB
Step 1/2 : FROM ubuntu:latest
 ---> 452a96d81c30
Step 2/2 : COPY test /
 ---> 2982811c946b
Successfully built 2982811c946b
Successfully tagged test:latest
```

test 镜像已经创建成功，进入/var/lib/docker/aufs 目录下查看镜像层的变化。

```
/var/lib/docker/aufs/diff# ls
47c4ed02b668dcf61970686e8b42d9b9e8c5b76d0e75acb0cbd5ad15dfc6b9f2
824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976
ed7b3b5ba6c11b0bd8182b9c940fc5d2402d4cdfbedb946a411366d091da0199
6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed
e85c04785e023817c8e6b64c2784ae931e7d7c7e84e0be9f5559a70a64db83bf
c41c662bb2e1bbb3ff0b729e8716135639de265580c5e240e04feb5357aa704b

/var/lib/docker/aufs/diff# cd c41c662bb2e1bbb3ff0b729e8716135639de265580c5e240e04feb5357aa704b
# ls
test
# cat test
hello world
```

可以看到只增加了一层，多了一个 test 文件，前 4 层被 ubuntu 和 test 两个镜像复用了，从而减少了磁盘的空间占用。

（3）AUFS 在 Container 中的使用

AUFS 在容器中的使用和在镜像中略有不同。在镜像中一个文件进行了改动，需要将整个文件进行复制，这样会对容器的性能产生一定的影响。在容器中，每一个镜像层最多只需要复制一次，后续的改动都会在复制的容器层上面进行，不需要再复制生成新的容器层。

首先在服务器上启动一个容器，查看文件层发生了哪些变化，如下所示。

```
/var/lib/docker/aufs/diff# ls
47c4ed02b668dcf61970686e8b42d9b9e8c5b76d0e75acb0cbd5ad15dfc6b9f2
824dcee16f4662ba1837912f9391a03a4336929af7667f0b0bed109a3a827976
ed7b3b5ba6c11b0bd8182b9c940fc5d2402d4cdfbedb946a411366d091da0199
6691558fa5744301fb7a9e65a058738ed1b22dc78c192e3d67be2b14a2af86ed
e85c04785e023817c8e6b64c2784ae931e7d7c7e84e0be9f5559a70a64db83bf
c41c662bb2e1bbb3ff0b729e8716135639de265580c5e240e04feb5357aa704b
af62bc6b680b5412febe50c3a1f5cf14e95b322aad38ea68fdd0e1b7c15dc6e7
af62bc6b680b5412febe50c3a1f5cf14e95b322aad38ea68fdd0e1b7c15dc6e7-init
```

发现多了两个文件层，其中的 init 文件层中主要存放与容器内环境相关的内容，这是一个只读文件层，另外一个文件层是一个可读写文件层，用来存储容器的写操作。当容器内部文件发生任何改变时，改变后的文件就会存储在这层文件系统中，当容器停止时它们仍然是存在的，只有容器被删除时才会删除这两个文件层。

同时还有一个/var/lib/docker/aufs/mnt 目录用来存放容器的 mount 目录，与/var/lib/docker/

aufs/diff 目录内容保持一致，当容器停止运行时，这些目录仍然是存在的，但是目录里面是空的，因为 AUFS 只在容器运行时才会把/var/lib/docker/aufs/diff 中的内容映射过来。

1.3 Docker 镜像及镜像仓库

1.3.1 什么是 Docker 镜像

Docker 镜像是由一层层的文件系统组成的特殊文件系统，为容器提供了运行时所需要的程序、配置、环境等。镜像属于只读文件系统，不能包含任何的动态数据，在构建完成后就不会被改变。

Docker 在运行容器前需要在本地存在对应的容器镜像，如果本地不存在该镜像，则 Docker 会先从镜像仓库中将对应的镜像拉取到本地，然后通过该镜像启动容器。

拉取完 Docker 镜像就可以查看本地是否已经存在对应的镜像，要查找本地拥有了哪些镜像可以通过 docker image ls 命令，列出本地存在的镜像，如下所示。

```
# docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test                latest              2982811c946b        7 days ago          79.6MB
nginx               latest              ae513a47849c        3 weeks ago         109MB
ubuntu              16.04               0b1edfbffd27        3 weeks ago         113MB
ubuntu              latest              452a96d81c30        3 weeks ago         79.6MB
```

如果想要删除某个镜像，可以通过 docke rmi[镜像:标签]命令实现。如下所示为删除 test 镜像：

```
# docker rmi test:latest
Untagged: test:latest
Deleted: sha256:2982811c946b4c5a76862e6f126cfa2368bb02281c38c7292aebf040d5edf729
Deleted: sha256:d83595d01aabad9d28689b8cdca72bbb05b7abbe558f64b760e9ff873dc792ca
```

这样就删除了 test:latest 镜像。其中的 latest 标签比较特殊，对于 Docker 镜像来说，如果不显式地指定标签，则默认会选择 latest 标签，表示最新版本的镜像。所以删除 test:latest 镜像也可以直接使用 docker rmi test，同样的拉取 test:latest 镜像也可以直接使用 docker pull test，省略 latest 标签。

1.3.2 构建 Docker 镜像

构建 Docker 镜像的方式有两种，一种是通过 docker commit 方式，另一种是通过 Dockerfile 文件构建的。相比之下，推荐使用 Dockerfile 的方式构建，因为 Dockerfile 方式简化了镜像的构建过程，并且可以更好地进行版本控制。首先简单讲解通过 docker commit 的方式构建镜像，然后会详细讲解如何通过 Dockerfile 的方式构建镜像。

1. 使用 docker commit 构建镜像

下面通过一个简单的示例讲解如何通过 docker commit 的方式构建镜像。

```
# docker run -it ubuntu:16.04 /bin/bash
root@2a87d7bbbbdf:/# apt-get -y update
root@2a87d7bbbbdf:/# apt-get -y install apache2
```

通过如上命令创建了一个基于 ubuntu:16.04 镜像的容器，并在容器中安装了 apache2。

```
# docker ps -a
CONTAINER ID  IMAGE         COMMAND      CREATED         STATUS PORTS                NAMES
2a87d7bbbbdf  ubuntu:16.04 "/bin/bash"  8 minutes ago    Exited (0) 7 seconds ago   silly_carson
```

可以看到生成的容器 ID 为 2a87d7bbbbdf，接着可以通过 docker commit 命令创建一个新的镜像。

```
# docker commit 2a87d7bbbbdf licheng17/apache2
sha256:d4c983bc07f5459cd6565ea55b27abd5f01a8c4ed06994f0fa747c3db02e9be9
# docker images
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
Licheng17/apache2     latest              d4c983bc07f5        About a minute ago   252MB
nginx                 latest              ae513a47849c        3 weeks ago          109MB
ubuntu                16.04               0b1edfbffd27        3 weeks ago          113MB
ubuntu                latest              452a96d81c30        3 weeks ago          79.6MB
```

成功地创建了一个新的镜像 licheng17/apache2，接着可以通过 docker push 命令把创建的镜像提交到镜像仓库中。在提交前需要先登录自己的 docker hub 账户，如果没有 docker hub 账户，则需要先注册。当然也可以创建自己的私有镜像仓库，登录私有仓库，将镜像上传。

```
# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID,
head over to https://hub.docker.com to create one.
Username: licheng17
Password:
Login Succeeded

# docker push licheng17/apache2
The push refers to repository [docker.io/licheng17/apache2]
e72a1f025462: Pushed
bf3d982208f5: Pushed
cd7b4cc1c2dd: Mounted from library/ubuntu
3a0404adc8bd: Mounted from library/ubuntu
82718dbf791d: Mounted from library/ubuntu
c8aa3ff3c3d3: Mounted from library/ubuntu
latest: digest: sha256:af8793d617dcc9f863ae02ac0f087963362ceddd3782087908bc4ab50fdca667 size:
1569
```

镜像上传成功后可以登录自己的 docker hub，查看是否增加了刚上传的 Docker 镜像 licheng17/apache2，如图 1-3 所示。

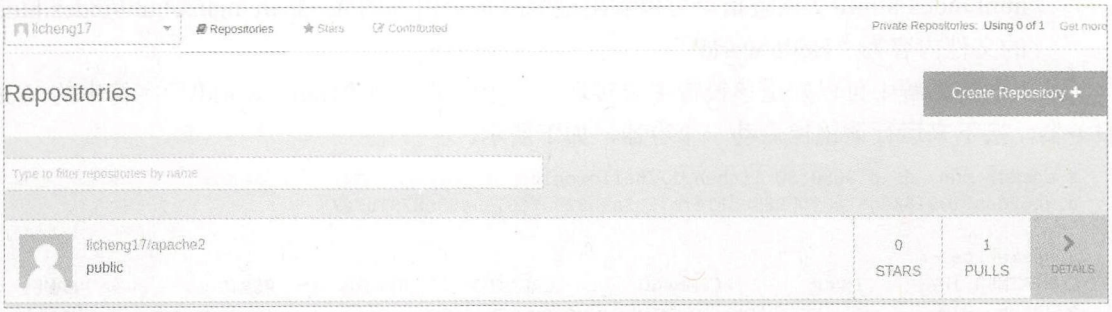


图 1-3 查看镜像

2. 使用 Dockerfile 构建镜像

下面详细讲解如何通过 Dockerfile 创建 Docker 镜像，相对 commit 的方式，Dockerfile 方式更加强大而灵活。

（1）如何使用 Dockerfile 构建镜像

要想通过 Dockerfile 方式构建 Docker 镜像，首先要有一个工作目录，然后在该目录中创建一个 Dockerfile 文件。在 Dockerfile 文件中添加编译指令，就可以通过 docker build 命令编译 Docker 镜像了，如下是一个简单的事例：

```
# mkdir imagedir
# cd imagedir/
```

```
~/imagedir# vim Dockerfile

~/imagedir# cat Dockerfile
FROM nginx:latest
MAINTAINER licheng17 "1076366950@qq.com"
RUN echo "Hello world!" > /usr/share/nginx/html/index.html

~/imagedir# docker build -t licheng17/helloworld .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM nginx:latest
 ---> ae513a47849c
Step 2/3 : MAINTAINER licheng17 "1076366950@qq.com"
 ---> Running in 3c998be9919c
Removing intermediate container 3c998be9919c
 ---> bf1af12040bd
Step 3/3 : RUN echo "Hello world!" > /usr/share/nginx/html/index.html
 ---> Running in ee0cc7a9fdb0
Removing intermediate container ee0cc7a9fdb0
 ---> 8587cdeb4df6
Successfully built 8587cdeb4df6
Successfully tagged licheng17/helloworld:latest
```

如上所示成功地创建了一个 licheng17/helloworld 镜像，上面的 Dockerfile 文件只有三行，具体含义如下。

- ROM nginx:latest：FROM 指令用于制订一个已经存在的镜像作为基础镜像，后续的所有指令都是基于该镜像运行的，所以在 Dockerfile 中的第一条指令都是 FROM 指令。在此 Dockerfile 中选择 nginx:latest 镜像作为基础镜像。
- MAINTAINER licheng17 "1076366950@qq.com"：这条指令并不是必须的，MAINTAINER 后主要用来表明该镜像的作者和作者邮箱信息，这条信息会被保存在容器内。
- RUN echo "Hello world!" > /usr/share/nginx/html/index.html：这条指令表明，在构建镜像时，会在镜像内部执行 RUN 后面的指令，即 echo "Hello world!" > /usr/share/nginx/html/index.html，从该条指令可以看出是修改镜像内部的/usr/share/nginx/html/index.html 的文件内容为“Hello world!”。

从上面的讲解中可以知道该镜像主要实现了一个能够返回“Hello world!”字符串的 Web 服务器，接下来运行该镜像启动一个容器，如下所示。

```
# docker run -d -p 8080:80 licheng17/helloworld
a72e626aa0109188dd57a61b5985e6f1efa32c3a36825af66136453b8781d7a7

# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                  NAMES
a72e626aa010        licheng17/helloworld   "nginx -g 'daemon of…"   15 seconds ago      Up 13
seconds       0.0.0.0:8080->80/tcp   elastic_ardinghelli

# curl localhost:8080
Hello world!
```

通过该镜像启动了一个容器，并将容器的 80 端口映射到本地的 8080 端口，然后通过 curl 命令请求本地的 8080 端口，正如预期所示返回“Hello world!”。

（2）Dockerfile 构建缓存机制

Docker 在构建镜像的过程中有一个缓存机制，当修改了 Dockerfile 中的某一行时，在构建的过程中，这一行的前几行将不会重复执行，而是直接从缓存中进行读取。这样可以缩短镜像的构建时间，但是在有些情况下，当希望能够强制忽略缓存时，可以在构建时强制添加

--no-cache 参数来实现，如下所示。

```
# docker build --no-cache -t licheng17/helloworld .
```

（3）Dockerfile 指令

在前面的例子中已经介绍了 FROM、MAINTAINER、RUN 指令，下面详细介绍 Dockerfile 中其他的常用指令。

- COPY 指令：该指令的作用是将构建上下文目录中的文件或目录复制到新的一层镜像内的目标位置。也就是将指定的本地环境中的文件或目录复制到镜像中指定的目录中。该指令和 cp 命令比较相似，cp 命令是从本地的一个目录复制到另一个目录，COPY 则是从本地的一个目录复制到镜像的指定目录中。如下例子所示。

```
COPY index.html /usr/share/nginx/html/
```

  如上指令将工作目录中的 index.html 目录复制到镜像的/usr/share/nginx/html/目录中，如果镜像中已经存在该文件，则会将文件进行替换。上面镜像中指定的目录是绝对目录，也可以指定镜像中工作目录的相对目录，而镜像的工作目录可以通过 WORKDIR 指令指定，后面会对 WORKDIR 进行讲解。

- ADD 指令：该指令实现的功能和 COPY 基本相同，但是比 COPY 增加了一些更高级的功能。比如在 ADD 中可以指定源目录为一个 URL，在这种情况下，Docker 会从这个链接中下载文件，然后再放到镜像的目标目录中。这与执行 wget 命令差不多，可以使用 RUN 指令来实现。当指定的本地源文件是一个 tar 压缩文件时，压缩格式为 gzip、bzip2 及 xz 的情况下，ADD 会自动将压缩文件解压到镜像对应的目标目录中去。例如在制作 Ubuntu 官方镜像中的 Dockerfile 中：

```
FROM scratch
ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /
...
```

  所以，当真的需要复制压缩文件，而不希望解压时不能使用 ADD 指令时，只能使用 COPY 指令来实现。同时需要注意 ADD 指令会令镜像构建缓存失效，从而可能会令镜像构建变得比较缓慢，所以在复制文件时推荐均使用 COPY，只有在需要自动解压文件时才使用 ADD 指令。

- CMD 指令：该指令可以指定容器启动时要运行的命令。例如，启动一个 Ubuntu 容器：

```
docker run -it ubuntu:16.04 /bin/bash
```

  其中，/bin/bash 表示启动容器时需要执行的命令，在 Dockerfile 中把该命令放在 CMD 后面，如下所示。

```
CMD ["/bin/bash"]
```

  这样在运行容器时，就不需要指明/bin/bash 命令，默认在启动时会执行 CMD 后指定的命令。CMD 还可以为命令添加如下参数：

```
CMD ["/bin/ls", "-l"]
```

  但需要注意 CMD 指定的命令可以被 docker run 指令命令覆盖。比如在 Dockerfile 中通过 CMD 指定了/bin/ls -l 命令，但是在运行容器时通过 docker run 指定了/bin/bash 命令，那么/bin/bash 就会把/bin/ls -l 覆盖掉，只会执行/bin/bash 而/bin/ls -l 并不会被执行。同时需要注意的是在 Dockerfile 中只能指定一条 CMD 指令，如果指定多条，只有最后一条 CMD 指令能生效。

- ENTRYPOINT 指令：ENTRYPOINT 与 CMD 命令实现的功能类似，但是 ENTRYPOINT

指定的命令在启动时并不能直接被覆盖。要覆盖 ENTRYPOINT 指定的命令则需要在 docker run 中通过--entrypoint 参数指定。

当指定 ENTRYPOINT 后，CMD 的内容就会作为参数传递给 ENTRYPOINT 指令。由于 CMD 容易被覆盖而 ENTRYPOINT 并不容易，这样在那些通常默认运行的可执行文件不发生变化，而参数在相对需要频繁变化的场景下就会非常有用。可以通过 CMD 指定一个默认的参数，而在运行时可以通过实际情况指定参数替换 CMD 中默认的参数。

- ENV 指令：该指令主要是为镜像设置环境变量。设置的环境变量在构建镜像的过程中可以被后续的指令所使用。ENV 配置环境变量的格式有两种：

```
ENV <key> <value>
ENV <key1>=<value1> <key2>=<value2>...
```

如下所示是一个 Dockerfile，通过该 Dockerfile 构建的 docker 镜像，可以通过修改环境变量 PRINT 构建不同的 Web 服务镜像，使返回不同的输出：

```
FROM nginx:latest
MAINTAINER licheng17 "1076366950@qq.com"
ENV PRINT "Hello world!"
RUN echo $PRINT > /usr/share/nginx/html/index.html
```

并且在启动该容器后，容器中仍然能够被使用该环境变量。如下所示，进入容器中，并输出该变量值。

```
# docker exec -it 62656f1ccbeb /bin/bash
root@62656f1ccbeb:/# echo $PRINT
Hello world!
```

在 Dockerfile 中通过 ENV 配置的环境变量仍然存在。

- ARG 指令：该指令与 ENV 类似，也可以配置环境变量，然后在构建镜像的过程中可以被后续的指令所使用。但是通过 ARG 配置的环境变量不能够被启动的容器所使用。ARG 配置环境变量的格式与 ENV 略有不同：

```
ARG <key>
ARG <key1>=<value1>
```

ARG 可以创建一个 key 但并不需要给它赋值，而是在 docker build 时用—build-arg <key>=<value> 指定该环境变量的值，也可以为 key 设置默认值，这样可以通过设置环境变量值的方式，仅使用一个 Dockerfile 就可以 build 出多个 Docker 镜像。在使用--build-arg 参数时需要注意，如果 Dockerfile 中没有声明对应的 key 值，使用--build-arg 赋值会报错。下面是一个例子，默认会 build 一个返回“Hello world!”的 Web 服务，但可以通过--build-arg 改变 PRINT 变量的值，而生成不同的镜像。

```
FROM nginx:latest
MAINTAINER licheng17 "1076366950@qq.com"
ARG PRINT="Hello world!"
RUN echo $PRINT > /usr/share/nginx/html/index.html
```

如上面的 Dockerfile，在 build 时使用--build-arg 参数：

```
# docker build -t licheng17/print1 --build-arg PRINT="I'm Licheng!" .
Sending build context to Docker daemon  3.072kB
Step 1/4 : FROM nginx:latest
 ---> ae513a47849c
Step 2/4 : MAINTAINER licheng17 "1076366950@qq.com"
 ---> Using cache
 ---> bf1af12040bd
Step 3/4 : ARG PRINT="Hello world!"
 ---> Using cache
```

```
 ---> 626b84f82ea4
Step 4/4 : RUN echo $PRINT > /usr/share/nginx/html/index.html
 ---> Running in ecb5c010d6b0
Removing intermediate container ecb5c010d6b0
 ---> 2874751d498d
Successfully built 2874751d498d
Successfully tagged licheng17/print1:latest

# docker run -d -p 8080:80 licheng17/print1
20518c80b37ba7cf5fa932f65ff400706bf5dda57653c9d85a8ac1ea242f958a
# curl localhost:8080
I'm Licheng!
```

在构建镜像的过程中，PRINT 确实被改变了。

- VOLUME 指令：该指令可以用来定义容器的匿名卷，通过卷可以使容器的目录绕过联合文件系统实现共享数据和数据持久化。在实际应用中一般需要将数据库等一些需要保存的动态数据文件保存在卷中。为了防止一些用户在运行时忘记将动态文件目录挂载为卷，在制作镜像时可以通过 Dockerfile 事先指定这些目录挂在为匿名卷，这样用户在使用时就不需要自己指定挂载。在容器启动时会默认将这些目录挂载到匿名卷中，直到容器被删除后这些卷才会被删除。如下在 Dockerfile 中创建一个匿名卷。

```
VOLUMN /data
```

这个/data 目录在运行时自动挂载一个匿名卷。也可以在启动容器时通过指定命名卷来挂载卷。

```
docker run -d -v data:/data xxxx
```

可以指定一个命名卷 data 作为容器/data 目录的挂载卷，在容器中的/data 中的所有写操作都会体现在该命名卷中。并且在删除容器后，该命名卷及其内部文件数据都是仍然存在的，并不会随容器一起删除，只能通过手动删除的方式删除命名卷。在运行新的容器时仍然可以挂载这个卷，这样就实现了数据的持久化功能。

- EXPOSE 指令：该指令用来声明容器暴露的端口，但是当运行容器时并不一定会开启指定的端口映射，但可以通过 docker run -P 的方式实现指定端口的随机映射，也就是说会在宿主机中随机启动一个端口映射指定的这个容器内部的端口。
- USER 指令：该指令用来指定当前用户，可以改变之后各层的当前用户身份，但需要注意的是 USER 只能够用来切换指定的用户，却不能创建用户，所以要切换的用户必须是事先建立好的，否则无法进行切换。
- WORKDIR 指令：该命令用来指定工作目录，以后各层的当前目录都会被改为该指定的目录，当该目录不存在时，WORKDIR 可以自动创建。在运行容器时可以通过-w 参数覆盖 WORKDIR 指定的工作目录。
- ONBUILD 指令：这个指令比较特殊，后面跟的是其他 Dockerfile 指令，这些指令不会在当前镜像构建的时候被执行，而是以当前镜像作为基础镜像构建下一级镜像的时候，后面跟的指令就会被触发执行。

1.3.3 搭建 Docker 镜像仓库

可以通过 Docker 官方提供的 registry 镜像非常容易地搭建一个镜像仓库。使用如下方式可以在本地搭建一个镜像仓库。

```
# docker run -d -p 5000:5000 registry
```

本地已经启动了一个镜像仓库。下面测试对该镜像仓库的使用，首先在本地创建一个标签

是 127.0.0.1:5000/licheng17/test 的镜像，接着可以将该镜像上传到本地的镜像仓库中，如下所示。

```
    # docker push 127.0.0.1:5000/licheng17/test
    The push refers to repository [127.0.0.1:5000/licheng17/test]
    89234cb5a7e6: Pushed
    7ab428981537: Pushed
    82b81d779f83: Pushed
    d626a8ad97a1: Pushed
    latest: digest: sha256:76d8604068c6c507b759f6e30e9a3519f835a6b04286a7f4cb2ad351df2e343f size:
1155
```

镜像上传成功，接着测试镜像拉取的功能，首先要把本地的镜像 127.0.0.1:5000/licheng17/test 删除，然后再拉取，如下所示。

```
    # docker rmi 127.0.0.1:5000/licheng17/test
    Untagged: 127.0.0.1:5000/licheng17/test:latest
    Untagged:
127.0.0.1:5000/licheng17/test@sha256:76d8604068c6c507b759f6e30e9a3519f835a6b04286a7f4cb2ad351df2
e343f
    # docker pull 127.0.0.1:5000/licheng17/test
    Using default tag: latest
    latest: Pulling from licheng17/test
    Digest: sha256:76d8604068c6c507b759f6e30e9a3519f835a6b04286a7f4cb2ad351df2e343f
    Status: Downloaded newer image for 127.0.0.1:5000/licheng17/test:latest
```

镜像拉取成功，说明镜像仓库工作正常。

1.4 Docker 网络

1.4.1 Docker 网络架构

从 Docker1.7 版本开始，Docker 已经将网络部分的代码单独分离出来，成为 Docker 网络库 Libnetwork。为了标准化网络的驱动开发步骤并支持多种网络驱动，Docker 通过 Libnetwork 实现了 CNM 网络模型。Docker 网络的整体架构如图 1-4 所示。

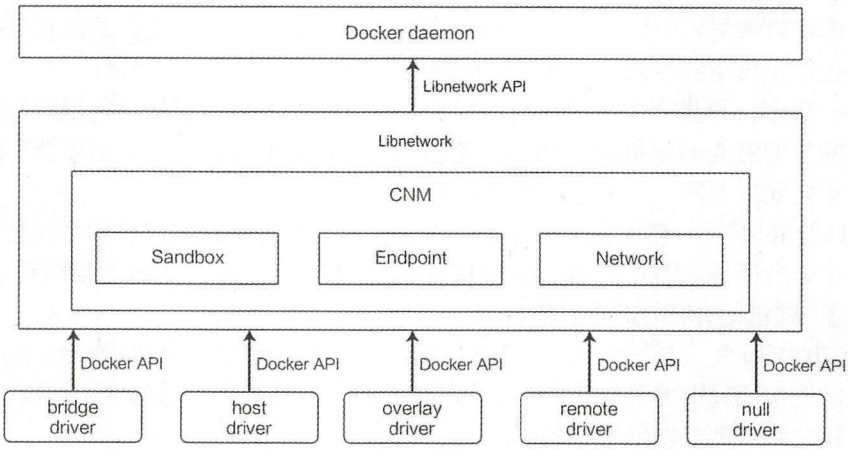


图 1-4 Docker 网络的整体架构

从图 1-4 中可以看出 Docker daemon 通过 Libnetwork 提供的 API 接口实现对网络的创建和管理功能。而 Libnetwork 通过 CNM 来实现这些网络功能，CNM 模型有三个组件：Sandbox（沙盒）、端点（Endpoint）、网络（Network），具体如下。

- Sandbox（沙盒）：每个沙盒包含一个容器网络栈（Network stack）的配置，配置包括容器的网口、路由表和 DNS 设置等。

- Endpoint（端点）：通过 Endpoint，Sandbox 可以被加入一个 Network 里。
- Network（网络）：一组能相互直接通信的 Endpoints。

上面的定义还是比较抽象的，理解起来比较困难，所以可以参考 CNM 模型在 Linux 上的实现技术来理解。比如，沙盒的实现可以是一个 Linux Network Namespace，Endpoint 可以是一对 VETH，Network 则可以用 Linux Bridge 或 Vxlan 实现。

Libnetwork 提供了 Bridge、Host、Overlay、Remote 和 Null 五种网络模式。

1）Bridge 驱动：此驱动为 Docker 的默认驱动，使用该驱动时，Libnetwork 将创建出来的 Docker 容器连接到 Docker 网桥上，能够满足容器的基本使用性需求。

2）Host 驱动：在该驱动模式下，Docker 容器直接使用宿主机的网络，与宿主机享有完全相同的网络环境。

3）Overlay 驱动：此驱动采用 VXLAN 中的 SDN controller 模式。在使用过程中，需要一个额外的配置存储服务，例如 Consul、etcd 和 Zookeeper 等，并在启动 Docker daemon 的时候通过参数指定所使用的配置存储服务地址。

4）Remote 驱动：该驱动实际上并没有实现 Docker 自己的网络服务，而是通过调用外部的网络驱动插件的形式实现网络服务。用户可以根据自己的需求选择或开发合适的网络插件。

5）Null 驱动：在这种网络模式下，Docker 仅仅为容器创建自己的 Network Namespace，提供网络隔离的功能，但并不会为容器创建任何网络配置，需要用户根据自己的需求为容器添加网络，并配置网络环境。

1.4.2 Docker 网络原理

这部分主要介绍 Docker 的网络原理，通过 Linux 系统模拟实现一个 Docker 默认网络的方法。

首先对 Docker 网络的实现原理进行整体简略的介绍，当 Docker 服务安装并启动后，在主机上输入 ifconfig 命令，可以发现主机上多了一个 docker0 的虚拟网桥，这个网桥为 Docker 的网络通信提供支持。如图 1-5 所示，在默认网络情况下，当启动一个 Docker 容器时，Docker 会为容器分配一个 IP 地址，并通过一对 veth pair 将容器的 eth0 绑定到主机的 docker0 网桥中。

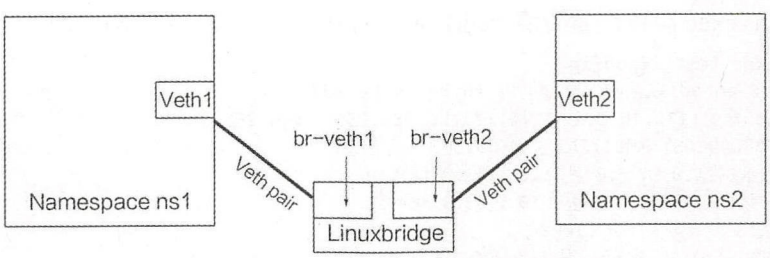


图 1-5 Docker 网络的实现原理

1. 模拟实现 Docker 网络

下面通过 Linux 系统模拟 Docker 网络模型，实现上面介绍的网络结构。

第一步：创建 Network Namespace

```
# ip netns add test
# ip netns list
Test
```

此时 Namespace test 创建成功了，在/var/run/netns 目录中可以看到一个 test 文件。当启动一个容器时，Docker 会为容器创建一个 Network Namespace，可以看到在/var/run/docker/netns

目录下生成了一个对应的文件。

第二步：添加网口到 Namespace

先创建 veth：

```
# ip link add veth0 type veth peer name veth1
```

在当前 Namespace 中可以看到 veth0 和 veth1：

```
# ip link list
... ...
11: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/ether f6:35:1a:b0:fd:03 brd ff:ff:ff:ff:ff:ff
12: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/ether a2:d9:ae:68:78:59 brd ff:ff:ff:ff:ff:ff
```

将 veth1 加到 namespace “test”：

```
# ip link set veth1 netns test
```

通过 ip link list 命令发现，当前 Namespapce 只能看到 veth0，而 veth1 已经找不到了。

```
# ip link list
... ...
12: veth0@if11: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default
qlen 1000
    link/ether a2:d9:ae:68:78:59 brd ff:ff:ff:ff:ff:ff link-netnsid 1
```

通过如下命令可以查看 test namespace 的网口，发现刚刚不见了的 veth1。说明 veth1 已经加入 test namespace 中。

```
# ip netns exec test ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
11: veth1@if12: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default
qlen 1000
    link/ether f6:35:1a:b0:fd:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

配置 Network Namespace 的网口。可以通过 ip netns exec 进行配置。

```
# ip netns exec test ifconfig veth1 172.16.0.2/16 up
```

通过 ip netns exec 可以配置查看 veth1 网口的 IP 地址正是 172.16.0.2。

```
# ip netns exec test ifconfig
veth1     Link encap:以太网  硬件地址 f6:35:1a:b0:fd:03
          inet 地址:172.16.0.2  广播:172.16.255.255  掩码:255.255.0.0
          UP BROADCAST MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:0 (0.0 B)  发送字节:0 (0.0 B)
```

这样一个隔离的容器网络就完成了，又该如何实现容器网络与外部的通信呢？这就需要用到 Bridge 了。

第三步：创建网桥

在默认的 Network Namespace 下创建 test0 网桥：

```
# brctl addbr test0
# brctl show
bridge name bridge id  STP enabled interfaces
test0  8000.000000000000 no
docker0  8000.024226c97719 no  veth917e72a
```

test0 网桥创建成功，其中 docker0 是 Docker 服务器自己的网桥。下一步是给 test0 分配 IP

地址并生效，充当 Gateway：

```
# ip addr add 172.16.0.1/16 dev test0
# ip link set dev test0 up
```

将 veth0“插到”test0 这个 Bridge 上：

```
# brctl addif test0 veth0
# sudo ip link set veth0 up
```

查看 test 网络生成了一条直连路由表，现在 test 网络可以 ping 通 test0 了，但由于没有默认路由，Docker 还不能与其他网络相通，通过 ping docker0 测试发现确实如此。

```
# ip netns exec test ip route show
172.16.0.0/16 dev veth1  proto kernel  scope link  src 172.16.0.2

# ip netns exec test ping -c 3 172.16.0.1
64 bytes from 172.16.0.1: icmp_seq=1 ttl=64 time=0.188 ms
64 bytes from 172.16.0.1: icmp_seq=2 ttl=64 time=0.085 ms
64 bytes from 172.16.0.1: icmp_seq=3 ttl=64 time=0.090 ms

--- 172.16.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.085/0.121/0.188/0.047 ms

# ip netns exec test ping -c 3 172.17.0.1
connect: Network is unreachable
```

为 test 网络添加一条默认路由表，测试 ping docker0 发现成功。

```
# ip netns exec test ip route add default via 172.16.0.1

# ip netns exec test ping -c 3 172.17.0.1
PING 172.17.0.1 (172.17.0.1) 56(84) bytes of data.
64 bytes from 172.17.0.1: icmp_seq=1 ttl=64 time=0.159 ms
64 bytes from 172.17.0.1: icmp_seq=2 ttl=64 time=0.143 ms
64 bytes from 172.17.0.1: icmp_seq=3 ttl=64 time=0.102 ms

--- 172.17.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2033ms
rtt min/avg/max/mdev = 0.102/0.134/0.159/0.027 ms
```

现在的 test 网络可以与主机相通了，但还不能访问外部网络，因为 icmp 包回来时找不到目的地，也就找不到 172.17.0.2 了，可以通过 iptables 来解决。

```
# ip netns exec test ping -c 3 114.114.114.114
PING 114.114.114.114 (114.114.114.114) 56(84) bytes of data.

--- 114.114.114.114 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2054ms

# iptables -t nat -A POSTROUTING -s 172.16.0.0/16 ! -o test0 -j MASQUERADE
# ip netns exec test ping -c 3 114.114.114.114
PING 114.114.114.114 (114.114.114.114) 56(84) bytes of data.
64 bytes from 114.114.114.114: icmp_seq=1 ttl=127 time=13.9 ms
64 bytes from 114.114.114.114: icmp_seq=2 ttl=127 time=48.7 ms
64 bytes from 114.114.114.114: icmp_seq=3 ttl=127 time=14.0 ms

--- 114.114.114.114 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 13.904/25.567/48.755/16.396 ms
```

可以看出添加完 iptables 规则后已经可以正常访问外网了。这样一个类 Docker 的网络就完成了，其实 Docker 的网络相对会比这复杂很多，但基本实现原理方法是一样的。

2. Docker 容器端口映射原理

上一节已经通过模拟实现 Docker 网络的方式讲解了 Docker 的网络原理。但还有一个重要的功能没有讲，就是容器端口映射的功能，即将容器的服务端口 a 绑定到宿主机的端口 b 上，最终达到一种效果：外部程序通过宿主机的端口 b 访问，就像直接访问 Docker 容器网络内部容器提供的服务一样。

如下通过 Docker 启动了一个 Nginx 容器服务，并将容器内部的 80 端口映射到本地主机的 9091 端口。然后可以通过本地的 9091 端口访问容器的 80 端口，即 Nginx 服务。

```
# docker run -d -p 9091:80 nginx:latest
d2bb56e4628cad43a42bd23f61fa2c6a1153c263bdc9ea00d2bb9dc165784162

# curl localhost:9091
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

容器是怎样实现端口映射功能的呢？在默认情况下，在 1.12.1 版本以前 Docker 引擎采用 docker-proxy 进程来实现，在之后的版本中，虽然增加了 iptables 的方式实现，但在默认情况下通过配置–userland-proxy=true 为每个 expose 端口的容器启动一个 proxy 实例来实现端口流量转发。docker-proxy 实际上是实现了在默认命名空间和容器命名空间之间的流量转发功能而已。

但是因为每增加一个端口映射，宿主机就会多出一个 docker-proxy 进程，一旦需要过多的端口映射，就需要增加过多的 docker-proxy 进程，这样将会消耗大量的资源。因此，在 1.7 及更高版本中，Docker 提供了一种完全由 iptables DNAT 实现的端口映射，但 docker-proxy 的方式依旧是默认方式，通过配置–userland-proxy=false 来选择 iptables DNAT 模式。

```
sudo iptables-save| grep 9091
-A DOCKER ! -i docker0 -p tcp -m tcp --dport 9091 -j DNAT --to-destination 172.17.0.2:80
```

从上述的 iptables 规则可以看出，iptables 将本地的 9091 端口通过 DNAT 映射到 172.12.0.2 的 80 端口，其中 172.12.0.2 就是 Nginx 的容器 IP。

本章作者：李成。

参考文献

[1] 陈显鹭. 自己动手写 Docker.[M].北京：电子工业出版社，2017.

[2] 理解 Docker 容器网络之 Linux Network Namespace：https://tonybai.com/2017/01/11/understanding-linux-network-namespace-for-docker-network/.

[3] “深入浅出”来解读 Docker 网络核心原理：http://blog.51cto.com/ganbing/2087598.

[4] 华为 Docker 实践小组. Docker 进阶与实战.[M].北京：机械工业出版社，2016.

[5] Docker 入门与实践：https://legacy.gitbook.com/book/hujb2000/docker-flow-evolution/details.

CHAPTER

2

第2章 Kubernetes 入门

本章主要围绕 Kubernetes 展开，首先介绍 Kubernetes 的发展历史、核心概念及优势；接着讲解 Kubernetes 的架构，并介绍多种安装方式；然后从一个例子着手，进一步分析如何使用 Kubernetes；接着介绍 Kubernetes 网络模型及原理，并介绍主流的开源网络组件；接着介绍 Kubernetes 的高级特性；最后围绕着 Kubernetes 生态进行讲解，主要包括 Helm 包管理、服务网格（Service Mesh）和无服务架构（Serverless）等。

本章的目的是帮助读者了解 Kubernetes 的发展，了解 Kubernetes 架构、基础概念、核心原理，以及如何在实际中使用 Kubernetes。无论开发人员、运维人员，还是 Kubernetes 的爱好者，相信都会有所收获。

2.1 Kubernetes 概述

Kubernetes 作为近几年火热的技术之一，受到了大量技术爱好者的关注。Kubernetes 也成了云计算领域中的一颗明星。本章分别从 Why、What、How 开始，介绍 Kubernetes 诞生的原因、Kubernetes 的发展历史、核心概念及它所带来的优势，帮助读者初步且完备地了解 Kubernetes。

2.1.1 什么是 Kubernetes

Kubernetes 官方文档给出了一个简短的解释："Kubernetes 是一个管理跨主机容器应用的开源系统，提供了应用的部署、维护及扩容缩容的基本机制。Kubernetes 项目由云原生计算基金会（CNCF）进行托管。"维基百科给出的解释是："Kubernetes（简称 k8s）用于提供容器应用的自动部署、扩容缩容以及管理容器应用，最开始由 Google 设计开发最后贡献给 CNCF 的开源容器集群管理项目。旨于提供管理跨主机的应用的部署、维护以及扩容缩容能力。Kubernetes 适用包括 Docker 在内的一系列容器工具。"

如何更通俗地理解 Kubernetes 是什么呢？其实，Kubernetes 是基于 Google Borg 系统超过 15 年的实践进行开源的分布式的容器编排管理平台，Kubernetes 旨在降低计算、网络和存储资源编排的复杂度，让运维人员和开发人员专注于以容器为载体的应用程序。Kubernetes 具备完善的集群管理能力，包括应用的快速部署，快速扩容缩容，跨主机调度，安全防护和准入机制，多租户资源隔离，透明的服务注册和服务发现机制，负载均衡、故障发现以及修复能力，多维度的资源配额管理能力。Kubernetes 还提供完善的管理工具，包括开发、部署测试、监控日志等各个环节。同时，Kubernetes 自身具备分布式、可扩展、可移植的特点。

1. 分布式平台

Kubernetes 是一个分布式平台，模块组件之间相互分离，各个模块各自负责一个功能，通过 Restful 及 gRPC 方式进行通信，模块之间保证了故障隔离。

2. 可扩展性

Kubernetes 具有各种扩展插件（Add-on），为 Kubernetes 提供了更加丰富的扩展功能。Kubernetes 也提供了可插拔式的调度模式。另外，Kubernetes 开放包括 CRI、CNI、CSI 等基础资源接口，在此基础上可以进一步对接支持，为 Kubernetes 提供更好的扩展性。

3. 可移植性

Kubernetes 平台自身不依赖于底层资源，可以运行在物理机、虚拟机和各类云主机中。CNCF 提出 Kubernetes 一致性模型，确保 Kubernetes 在不同云环境的一致性；同时，Kubernetes 也集成了各大云厂商的 Cloud Provider，它可以帮助 Kubernetes 在对应的公有云环境上创建公有云上的资源。

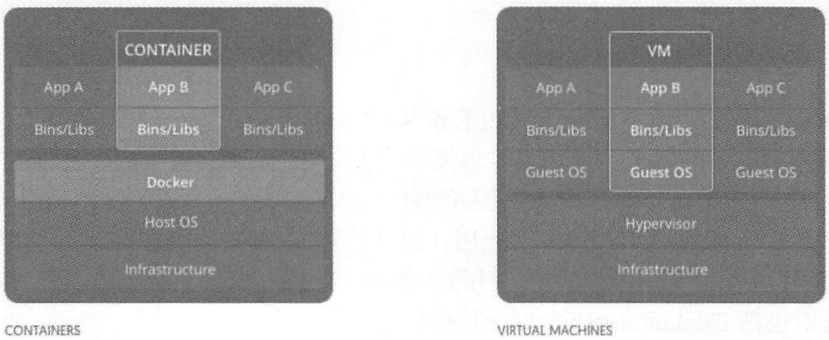
2.1.2 为什么选择 Kubernetes

说到为什么选择 Kubernetes，就不得不提容器。

回顾一下云计算的历史。1959 年，学术中提出了虚拟化的概念，虚拟化是云计算基础架构的基石。之后，云计算始终只是停留在学术层面上。直到 1998 年，VMware 成立并首次引入 X86 的虚拟技术，真正迎来了云计算的时代。亚马逊云计算服务于 2006 年推出，在随后的

十年间，它可以说趁着云计算趋势改变了整个IT行业。2010年，开源云计算IaaS平台OpenStack发布，在多家主流公司的推动下迎来了爆发，越来越多的企业利用OpenStack搭建公有云和私有云平台。2013年，dotCloud公司开源了容器引擎Docker，相比于虚拟机，Docker的秒级启动、环境标准化、轻量级、资源利用率高、快速扩缩容等优势立刻引起了行业的关注。

图2-1对比了容器方式（a）和传统虚拟化方式（b）的差别。容器是在操作系统层面的虚拟化，它复用了本地主机的操作系统；而传统虚拟化只是在硬件资源层面的虚拟化。



（a）容器方式　　（b）传统虚拟化方式

图2-1　容器方式和传统虚拟化方式的差别

“容器的出现带来了很多好处，而在容器的实践过程中，容器的编排、管理平台变得更为迫切，关注程度也大大超过了容器本身。主流的容器编排、管理平台有：Kubernetes、Swarm、Mesos。Swarm是Docker的原生编排工具，Kubernetes 和Swarm都是基于Golang语言进行开发的，Mesos则采用C++语言开发。图2-2是GitHub中三种编排管理工具的Star和Fork数量（数据2018.03采集）。”

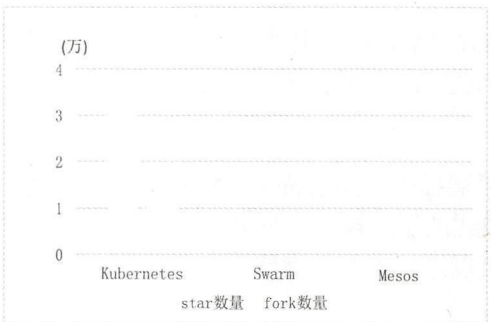


图2-2　GitHub中三种编排管理工具的Star和Fork数量

可以看出Kubernetes在Star和Fork数量上明显领先于其他两者。2017年，Docker公司在巨大的争议中终于完成了向商业公司的重要转变，由Kubernetes社区所主导的、全新的容器生态正式拉开序幕，也宣告了Kubernetes已经在容器编排管理工具的角逐战中胜出。

使用Kubernetes究竟有什么好处呢？

- 拥抱微服务。微服务架构将巨大单体式应用分解为多个服务，每个服务通过RPC或者API进行通信，具备了各个自服务容易开发、维护的优点，另外子服务还具备独立部署、快速扩展等优点。Kubernetes对微服务本身有很好的支持，应用本身通过Deployment进行部署，各个服务运行在Pod中，Pod之间服务通过Service具备的服务发现实现相互间的通信。同时微服务架构也恰好是云原生应用的一种体现。

- 容器编排。Kubernetes 帮助使用者通过简单易用的 API 高效地管理成千上万个运行在容器中的微服务。Kubernetes 在 1.6 版本中已经支持 5000 个节点，这也说明 Kubernetes 具备大规模集群的编排管理能力。同时，Kubernetes 还具备包括监控、日志、包管理等各种完善而专业的工具链，这将大大减轻运维和开发人员的负担，传统动辄十几人的团队，采用 Kubernetes 方案之后只需精湛的小团队就能应付自如。
- 服务注册发现。微服务的实践过程中存在各种服务依赖关系，因此服务的注册发现十分重要。Kubernetes 对服务进行抽象，通过抽象的服务层动态地解析到对应的容器服务。Kubernetes 同时提供了 DNS 和环境变量两种方式，帮助实现服务的注册和发现，Kubernetes 早期版本中使用的环境变量方式实现，现在 Kubernetes 则默认使用 DNS，通过使用 DNS 将服务名称解析为服务的 IP 地址，然后 Proxy 转到对应的 Pod。

2.1.3 Kubernetes 基本概念

Kubernetes 定义了资源对象，资源对象是一种逻辑实体，使用者需要使用 kubectl 命令管理工具或者直接通过 Kubernetes API 进行调用实现对资源对象的创建、修改或删除的操作。同时，将资源对象的状态保存到 etcd 中。Kubernetes 主要资源对象有 Pod、Service、ReplicaSet、Deployment、StatefulSet 和 PersistentVolume 等。

1. Pod

Pod 是运行在 Kubernetes 集群上创建部署的最简单的构建单位。Pod 对一个或者多个应用程序的容器进行了封装，在 Pod 中，容器共享容器运行时所需要的网络与存储资源。一个 Pod 表示由一个容器或者一组耦合容器共享资源的 Kubernetes 应用实例。

网络与存储资源的共享：每一个 Pod 都被分配了一个唯一的 IP 地址。Pod 里的所有容器共享着一个网络空间，这个网络空间包含了 IP 地址和网络端口。Pod 内部的容器彼此之间可以通过 localhost 相互通信；Pod 可以指定一系列的共享存储卷。Pod 里所有的容器都有权限访问共享卷，同时可以使用共享卷分享数据。

Pod 内部结构如图 2-3 所示。

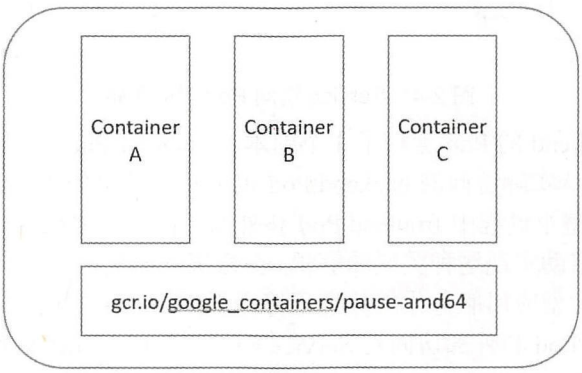


图 2-3 Pod 内部结构

图 2-3 中，Pause 容器被称为 Pod 的根容器，根容器的作用主要有两点：Pause 状态代表了 Pod 的状态；Pod 中其他容器共享 Pause 容器的网络资源和存储资源，实现了 Pod 中容器资源共享的问题。

下面是 Kubernetes 中 Pod 资源定义的例子：

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
imagePullPolicy: Always #三个选择 Always、Never、IfNotPresent
    ports:
    - containerPort: 80
      protocol: HTTP
```

kind:Pod 表明定义了一个 Pod 资源对象。metadata.name 为 Pod 的名字。这里声明了一个 app=nginx 的标签。Pod 中包含了一个 name=nginx，镜像为 nginx:latest 的容器，imagePullPolicy 指明拉取镜像的规则：Always 为每次都重新拉取镜像，Never 为每次不管本地是否存在都不拉取镜像，IfNotPresent 为如果本地存在就不拉取而如果没有就拉取镜像。Ports 指明了容器需要暴露的端口及协议。

2. Service

Kubernetes Service 定义了这样一种抽象：通过 Service 资源可以访问相应的 Pod 资源，Pod 资源与 Service 的对应关系则通过 Label Selector 关联实现。Service 在 Kubernetes 中十分重要，通过 Service 进行 Pod 之间的访问以及外部到 Pod 的访问。图 2-4 体现了 Service 访问 Pod 时的逻辑。

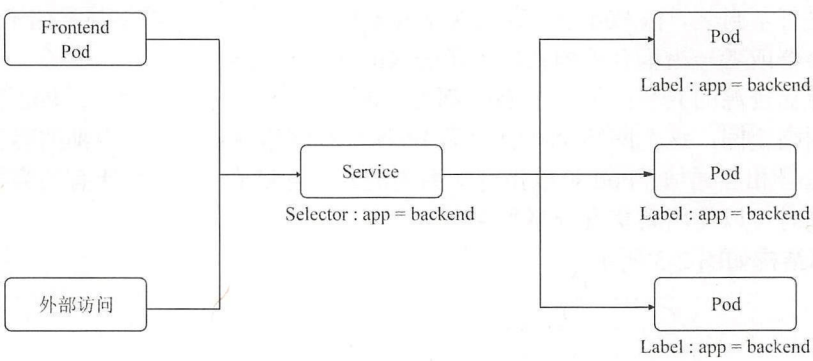


图 2-4　Service 访问 Pod 时的逻辑

可以看出 app=backend 的 Pod 运行了 3 个副本。frontend Pod 或外部访问不用关心具体调用了哪个 backend 副本。实际访问的 backend Pod 也可能会发生变化，同样不需要 frontend Pod 或者外部访问知道，在整个过程中 frontend Pod 和外部访问不需要跟踪这一组 backend 的状态。Service 对象定义能够帮助实现这种关联的解耦。在微服务架构中，每一个服务对应 Kubernetes 中的一个 Service，将大型应用服务化拆分也正符合 Kubernetes 的设计理念。

如何实现 frontend Pod 和外部访问对 Service 的访问？Kubernetes 为每一个 Service 分配了一个全局唯一的虚拟 IP，成为 Cluster IP，这样就可以基于传统的网络通信方式访问 Service。

下面在之前的 Pod 资源上创建 Service：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  ports:
    - port: 80
```

```
  selector:
    app: nginx
```

上面定义了一个名为 nginx-service 的 Service 资源对象，Service 暴露的端口是 80，通过 selector 中 app=nginx 指定 Service 的 backend 是前文创建的 nginx Pod。

3. ReplicaSet

ReplicaSet（RS）是对 Replication Controller 在 Selector 功能上的升级，ReplicaSet 支持集群选择器的功能，而 Replication Controller 只支持等号选择器的功能。ReplicaSet 的使用场景是确保 Pod 在任何时候都能保持指定的副本数，图 2-5 所示为 ReplicaSet 的逻辑。

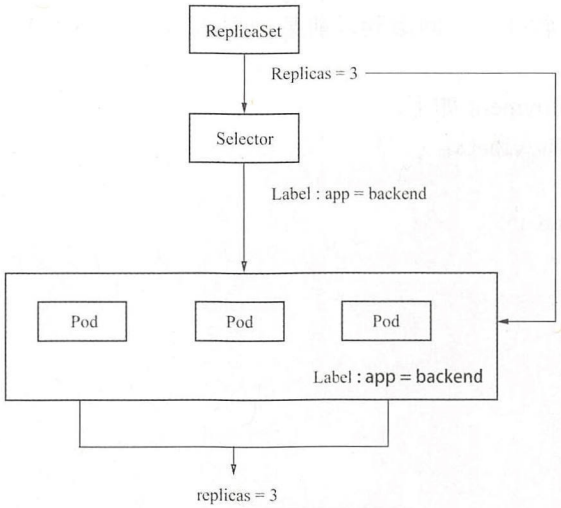


图 2-5　ReplicaSet 的逻辑

可以看出，当 Pod 数量小于 ReplicaSet 指定的副本数的时候，就会新起 Pod，直到达到副本数；ReplicaSet 与 Pod 的关联方式也与之前 Service 与 Pod 关联的方式相似，通过 Selector 实现。

下面创建 ReplicaSet 资源，ReplicaSet 中包含了之前名为 nginx Pod 的完整定义。

```
apiVersion: v1
kind: ReplicaSet
metadata:
  name: nginx-rs
spec:
  replicas: 3
selector:
  matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
spec:
      containers:
      - name: nginx
        image: nginx:latest
imagePullPolicy: Always
        ports:
        - containerPort: 80
          protocol: HTTP
```

上面定义了一个名为 nginx-rs 的 ReplicaSet 资源，replicas=3 指定副本数为 3。template 内

容即上文中 nginx Pod 的定义。ReplicaSet 通过 Selector 实现对 Pod 的控制，Selector 支持 matchLabels 和 matchExpressions 两种方式。同时，可以修改 replicas 的数量实现 Pod 的扩容或缩容功能，修改 template 中 image 镜像可以实现滚动升级的功能。

4. Deployment

Deployment 的主要功能是为了提供更好的编排、部署的支持，在 Kubernetes v1.2 版本开始引入，它是在 ReplicaSet 基础上实现的。和 ReplicaSet 相似，根据 Deployment 中描述期望的集群状态，逐步更新成期望的集群状态。Deployment 的主要职责同样是为了保证 Pod 的数量和健康。相比 ReplicaSet 的不同点，Deployment 提供了上线部署、滚动升级、创建副本、暂停上线任务、恢复上线任务、回滚到以前某一版本等功能，Deployment 大大降低了部署的复杂度与风险性。

创建 Nginx 的 Deployment 如下。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
selector:
  matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
spec:
      containers:
      - name: nginx
        image: nginx:latest
          imagePullPolicy: Always
        ports:
        - containerPort: 80
          protocol: HTTP
```

上面 Deployment 资源的定义与 ReplicaSet 的定义基本类似。主要的差别在于 API 版本的声明不同及资源类型不同：apiVersion: extensions/v1beta1，kind: Deployment。

5. StatefulSet

ReplicaSet 和 Deployment 主要针对的是无状态应用的部署，对于有状态并不能很好地支持。在 Kubernetes v1.5 中引入了 StatefulSet 的概念，并在 v1.9 版本中 stable、StatefulSet 是对 PetSet 的升级，用于对有状态应用的支持。StatefulSet 具备以下特性：

- StatefulSet 创建的 Pod 具备稳定、唯一的网络标识。
- StatefulSet 创建的 Pod 具备稳定的持久化存储，同时 Pod 的删除不会级联删除数据卷。
- 在部署和扩容的过程中，Pod 是逐个进行创建的，创建下一个 Pod 之前需要保证之前的 Pod 是 ready 状态；同样，删除的过程也是逐个进行删除的，最后创建的 Pod 将最先被删除。

下面以创建 MySQL 为例，代码如下。

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
  labels:
```

```
    app: mysql
spec:
  ports:
  - port: 3306
  clusterIP: None
  selector:
    app: mysql
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: mysql-statefulset
spec:
  serviceName: "mysql-service"
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: mysql
        image: mysql:5.6
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: db
          mountPath: /data/db
  volumeClaimTemplates:
  - metadata:
      name: db
      annotations:
        volume.beta.kubernetes.io/storage-class: "fast"
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

上面定义了两个资源。一个是名为 mysql-service 的 Headless Service，一个是名为 mysql-statefulset 的 StatefulSet 资源。Headless Service 是 clusterIP=None 的 Service，对定义了 selector 的 Headless Service，Endpoint 控制器创建了直接到达 Service 后端 Pod 的 Endpoints 记录，这就实现了 StatefulSet 创建的 Pod 具备稳定、唯一的网络标识。volumeClaimTemplates 表明使用存储供应商提供的持久化存储卷为 MySQL 应用提供稳定的持久化存储。

6. PersistentVolume

PersistentVolume（持久化存储，简称 PV）资源是 Kubernetes 对存储的抽象。研发人员或运维人员只需关心如何通过 PV 提供持久化存储功能而无须关注具体的存储底层的实现，PV 在 Kubernetes 中是独立于 Pod 存在的，那么在 Kubernetes 中如何为 Pod 提供数据卷呢？Kubernetes 提供了另一种资源 PersistentVolumeClaim（持久化存储声明，简称 PVC），PVC 允许 Pod 声明指定大小和指定访问模式的方式使用 PV 资源。

访问模式包括：

- ReadWriteOnce——卷可以被单个 Pod 以读写模式挂载。
- ReadOnlyMany——卷可以被多个 Pod 以只读模式挂载。
- ReadWriteMany——卷可以被多个 Pod 以读写模式挂载。

Kubernetes 中 PV 支持的存储底层类型为插件的形式，目前支持的插件有：GCEPersistentDisk、

AWSElasticBlockStore、AzureFile、AzureDisk、FC（Fibre Channel）、FlexVolume、Flocker、NFS、iSCSI、RBD（Ceph Block Device）、CephFS、Cinder（OpenStack block storage）、Glusterfs、VsphereVolume、Quobyte Volumes、HostPath、VMware Photon、Portworx Volumes、ScaleIO Volumes 和 StorageOS。主要基于各大云厂商、存储厂商以及开源的存储方案。其中，HostPath 使用的宿主机的存储能力主要用于在单节点的场景。

下面以 HostPath 为例创建一个 PV 资源。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: hostpath-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
path: /tmp
```

上述创建了一个名为 hostpath-pv 的 PV 资源，storage：5Gi 表明 PV 的大小是 5GB，访问模式是 ReadWriteOnce，采用的是 hostPath 方案，对应宿主机的目录为/tmp。如果想要 Pod 使用刚刚创建的 PV 资源，需要再定义一个 PVC 资源。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: hostpath-pvc
spec:
  resources:
    requests:
      storage: 5Gi
  accessModes:
    - ReadWriteOnce
```

然后在 Pod 资源定义对上述创建的 PVC 进行引用。

```
volumes:
- name: myvolume
  persistentVolumeClaim:
    claimName: hostpath-pvc
```

2.2 Kubernetes 架构及安装

Kubernetes 是一个分布式架构系统，各个组件各自负责一个功能，可以进行独立部署。Kubernetes 也支持在各种平台环境下进行部署，并保证系统的一致性。本部分将首先介绍 Kubernetes 的架构及各个组件的功能，然后再讲解 Kubernetes 完整的安装过程。

2.2.1 Kubernetes 架构

Kubernetes 主要分为两个节点类型：master 和 node（Kubernetes 较早版本中称为 Minion）。master 作为集群的控制节点，主要负责集群的控制、管理功能，以及负责接收对集群资源操作的命令；node 作为集群的工作节点，实际工作的 Pod 将运行在这类节点上。图 2-6 所示为 Kubernetes 的具体架构。

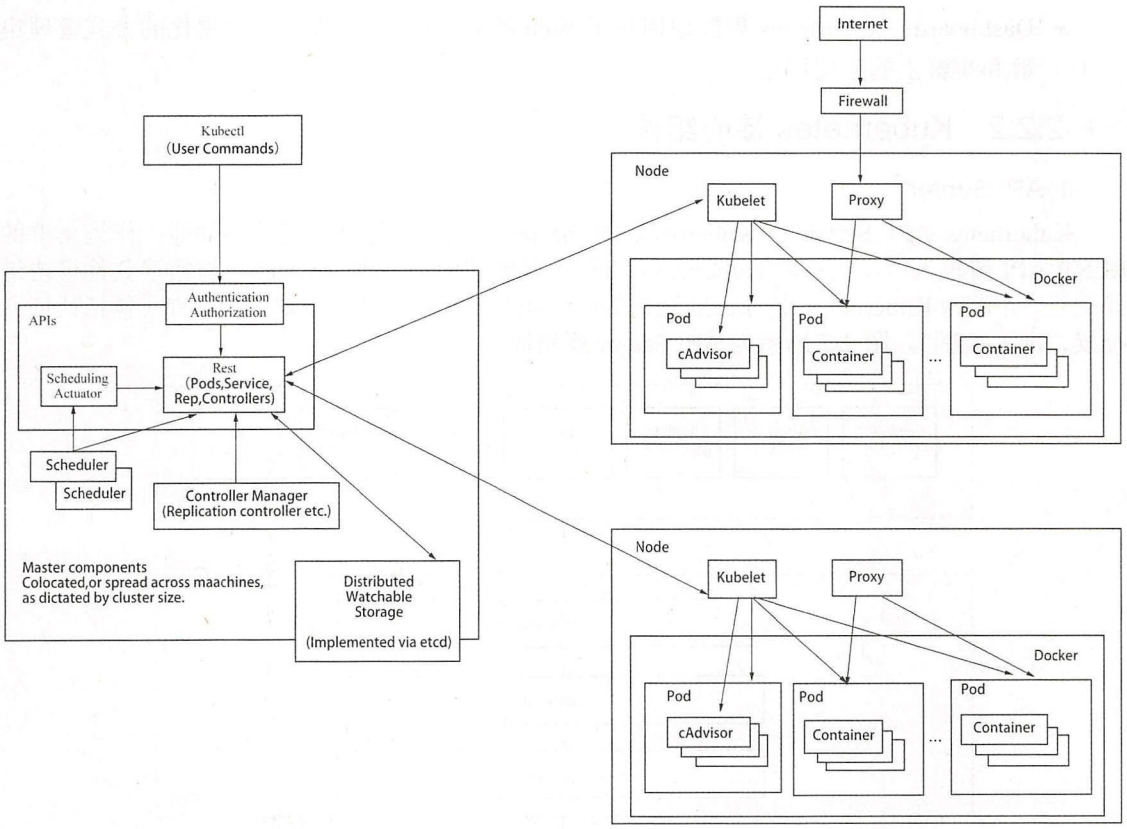


图 2-6 Kubernetes 的具体架构

可以看出 Kubernetes 主要由以下核心组件组成。

- API Server：提供了各类资源对象的增删改查及 watch 的 Restful 接口，同时负责模块间的相互通信，并提供认证、授权、访问控制、API 注册和发现等机制。
- Controller Manager：作为集群的控制中心，负责维护集群的状态达到期望的状态；Controller Manager 包含了 replication controller、endpoints controller、namespace controller 和 serviceaccounts controller 等，这些 controller 分别负责控制一块。
- Scheduler：负责资源的调度，根据集群的资源使用情况，将 Pod 调度某个节点；同时，Scheduler 组件是可插拔的，可以根据自身的业务设计调度策略。
- kubelet：运行在集群中的每一个节点上的 Agent，负责维护容器的生命周期。
- Container Runtime：负责管理容器的真正运行以及容器镜像的管理，Kubernetes 支持的 Container Runtime 有 Docker、Rkt、RunC 及符合 OCI 标准的实现。
- kube-proxy：为 Service 提供服务发现和负载均衡能力。
- etcd：保存集群中配置、资源状态等信息，通过 watch 操作，其他组件可以快速地获取资源的变更。

除了核心组件，还有一些插件在 Kubernetes 中提供了非常重要的功能。

- DNS：负责为整个集群提供 DNS 服务，在 Kubernetes 中承担着服务发现的作用。
- Heapster：对集群资源进行监控。

- Dashboard：Kubernetes 集群通用基于 Web 的 UI 界面，允许用户可视化的方式管理集群和集群上的应用程序。

2.2.2 Kubernetes 核心组件

1. API Server

Kubernetes API Server 在 Kubernetes 的 master 节点启动运行，其核心功能：作为集群的 REST API 请求入口，支持对 Kubernetes 的资源对象的增删改查及 watch；与集群其他模块进行交互，并作为 Kubernetes 中与 etcd 通信的唯一组件；拥有完备的集群安全机制（包括认证、授权、准入控制）。图 2-7 所示为 API Server 逻辑图。

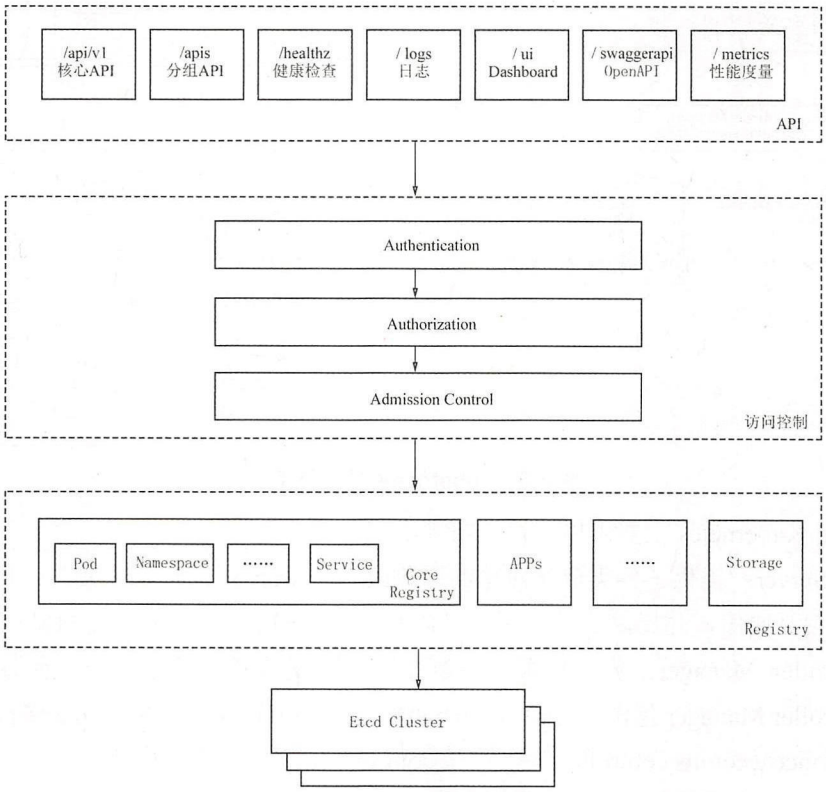


图 2-7 API Server 逻辑图

图中 Registry 是真正负责 etcd 存储的逻辑部分，实际包含了 ControllerRegistry、EndpointRegistry、NodeRegistry、PodRegistry 和 ServiceRegistry 等接口。另外，Kubernetes 资源对象都有一个 resourceVersion 作为其元数据，API Server 借此保证对资源对象操作的原子性。resourceVersion 是用于标志一个资源对象的内部版本的一个字符串，客户端可以通过它来判断该对象是否被更新过。

API Server 通过 kube-apiserver 进程提供服务，并提供了多种访问方式。

- 本地端口。使用 HTTP 访问，默认端口为 8080。该端口是一个非安全端口，没有用户认证和授权检查机制。如果需要自定义该端口，可以通过 API Server 的启动参数“--insecure-port”的值修改默认值；默认的 IP 地址为“localhost”，可以通过启动参数“--insecure-bind-address”的值修改该 IP 地址。该端口最初是为 Kubernetes 的其他组件提供 API 请求以及用于测试场景的。

- 安全端口。使用 HTTPS 访问，默认端口为 6443，基于 Token 文件或客户端证书及 HTTP Base 的认证。在 API Server 启动时分别通过--tls-cert-file 参数传入证书文件和--tls-private-key-file 参数传入私有密钥文件。如果需要自定义该端口，则在启动时传入—secure-port 参数。如果需要自定义绑定的网络接口地址，则在 API Server 启动时传入--bind-addrerss 参数。

2. Controller Manager

Controller Manager 由 kube-controller-manager 和 cloud-controller-manager 两部分组成，是 Kubernetes 集群的控制中心，它通过 API Server 监控整个集群的状态，并确保集群的状态达到期望的状态。cloud-controller-manager 是在 Kubernetes v1.6 中引入，在 Kubernetes 启用云服务商的 Cloud Provider 的时候才需要，使用云服务商的 Controller Manager 进行控制。这里主要以 kube-controller-manager 进行讲解。

kube-controller-manager 分为默认启动的 Controller 和默认不启动的 Controller。

1）默认启动的 Controller。

- EndpointController。
- ReplicationController。
- PodGCController。
- ResourceQuotaController。
- NamespaceController。
- ServiceAccountController。
- GarbageCollectorController。
- DaemonSetController。
- JobController。
- DeploymentController。
- ReplicaSetController。
- HPAController。
- DisruptionController。
- StatefulSetController。
- CronJobController。
- CSRSigningController。
- CSRApprovingController。
- TTLController。
- TokenController。
- NodeController。
- ServiceController。
- RouteController。
- PVBinderController。
- AttachDetachController。

2）默认不启动的 Controller。

- bootstrapsigner。
- tokencleaner。

其中，Replication Controller 的作用是确保集群中一个 Replication Controller 关联的 Pod 都

保持一定的副本数处于正常运行状态，主要使用在以下场景：确保 Pod 数量，以保证高可用要求；系统扩容和缩容；滚动更新。只有当 Pod 的重启策略 RestartPolicy=Always 时，Replication Controller 才会管理 Pod 的例如创建、销毁、重启等操作。

Node Controller 通过 API Server 发现和监控集群中的各个 Node 节点以及管理各个 Node 节点的相关控制功能。每个 Node 节点上都会部署 Kubelet 组件，Kubelet 在启动时向 API Server 注册 Node 节点信息，并定时向 API Server 发送节点信息。Node 节点具有三种状态：True、False、Unknown。当 Node 节点状态为 False 和 Unknown 时，则该节点将进入待删除队列，该节点上的 Pod 也将进入驱逐队列。

ResourceQuota Controller 的作用是对集群资源对象进行资源的配额管理，确保资源对象不会超量占用系统资源，从而避免了由于某些应用运行过程中资源占用过大导致对其他应用服务造成的影响，同时也可能对集群中关键服务造成影响，从而对整个集群的平稳运行造成影响。

Kubernetes 主要有三个维度的资源配额控制：容器，可以对 CPU 和 Memory 进行限制；POD，可以对一个 Pod 整体以及包含的所有容器的资源进行限制；Namespace，为一个命名空间下的资源进行限制。

在配置 Pod 时具有两类的资源配置项，Requests 表示资源的请求，Limits 表示资源的限额，这两类配置只能针对 Pod 中容器的 CPU、Memory 进行配置。LimitRange 则在 Pod 和容器层面都提供了 CPU、Memory 的配额限制。ResourceQuotas 针对 Namespace 层面进行资源配额管理，ResourceQuotas 支持的计算存储资源包括 CPU、Memory、Storage，并且也提供了 Congfigmaps、Pods、Services 等各类资源数量的配额管理。

Namespace Controller 通过 API Server 定时读取 Namespace 的信息，当 Namespace 的状态为 Terminating 时，则对该 Namespace 进行删除。同时，也会删除该 Namespace 下的 Deployment、Service、ServiceAccount、ReplicaSet、Pod、Secret、ResourceQuota 等资源。在这个过程中，Admission Controller 会通过 namespacelifecycle 阻止在 Namespace 中创建新的资源。

3. Scheduler

kube-scheduler 是 Kubernetes 的核心组件之一，具有可插拔、独立进程启动的特点。它负责分配调度 Pod 到集群内的节点上，通过监听 kube-apiserver，查询还未分配 Node 的 Pod 以及集群中所有节点的资源使用情况，然后根据调度策略将这些 Pod 绑定到优先级最高的可用节点上。

1）kube-scheduler 的调度流程。

- 可行性检查：在集群中找到一批可以被调度的节点，又称为预选过程（Predicte）。
- 计分：计算上一步骤得到的节点的分数，又称优选过程（Priority）。
- 对最高分的节点进行轮询算法选择，得到唯一被调度的节点。

图 2-8 所示为 kube-scheduler 的调度过程。在具体实践中，Kubernetes 内置了丰富的调度策略，默认具备指定节点、Taints 和 Tolerations（污点和容忍）、Pod 优先级调度等策略。

2）指定 Node 节点。有三种方式指定 Pod 只运行在指定的 Node 节点上：

- nodeSelector：节点标签选择，Pod 将只调度到匹配指定 label 的 Node 节点上，作用在预选过程。
- nodeAffinity：节点亲和性，支持更为丰富的集合操作。nodeAffinity 目前支持 requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnored-DuringExecution 两种方式，前者作用在预选过程代表必须满足条件，后者作用在优选过程代表优选条件。

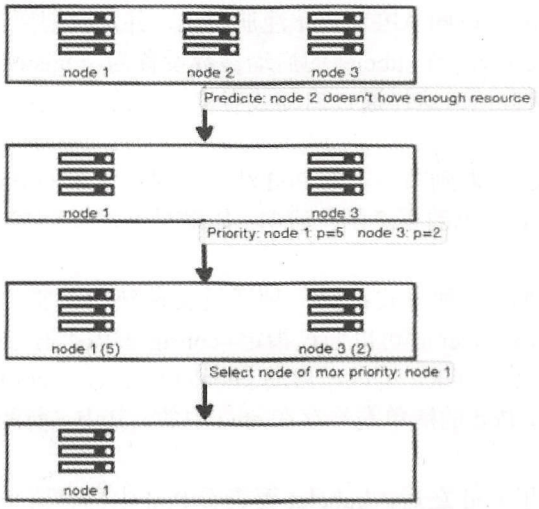


图 2-8 kube-scheduler 的调度过程

- podAffinity：Pod 亲和性，调度到满足条件的 Pod 所在的 Node 上。podAffinit 也同样支持 requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnored-DuringExecution 两种方式，用于预选和优选过程。

3）Taints 和 Tolerations（污点和容忍）。Taints 和 Tolerations 通常会一起使用，目的是确保 Pod 不会被调度到不正确的节点上，Taint 作用于 Node，而 Toleration 则作用于 Pod。通过给节点设置为 Taints，可以标志出这些节点不接受任何 Pod，所以 Kubernetes 不会将 Pod 调度到这个节点上。也可以通过给 Pod 设置为 Tolerations，让这些 Pod 部署到能够容忍标记为 Taints 的节点上。Taint 支持以下三种类型：

- NoSchedule：新创建的 Pod 不会调度到该节点上，不影响原先正在运行的 Pod。
- PreferNoSchedule：不影响原先正在运行的 Pod，新创建的 Pod 优先不会调度到该节点上，是 NoSchedule 的一种软限制。
- NoExecute：新创建的 Pod 不会调度到该节点上，并且对于已在运行的 Pod，如果设置了 tolerationSeconds，则经过 tolerationSeconds 时间 Pod 会被驱逐。

4）Pod 优先级调度。Kubernetes v1.8 版本引入了 Pod 优先级调度（抢占式调度），kube-scheduler 会根据 Pod 的优先级进行调度。优先级表明了 Pod 相对于其他 Pod 的重要程度，开启 Pod 优先级特性后，Kubernetes 会优先保证具有高优先级的 Pod 能被成功调度和运行。在调度过程中，当高优先级的 Pod 调度不成功时，低优先级的 Pod 会被驱逐，节点上空出更多资源保证高优先级的 Pod 的使用。在 v1.9 及以后的版本，优先级同时会影响调度的顺序以及资源不足时 Pod 被驱逐的顺序。当 Pod 完成调度之后，修改或删除 Pod 的优先级资源，不会对原有的 Pod 造成影响。

4. kubelet

在 Kubernetes 集群中的每一个节点上会运行一个 agent 进程：kubelet，该进程默认监听 10250 端口。当 kube-scheduler 将 Pod 调度到某个节点时，Master 节点就会向该节点的 kubelet 下发 Pod 具体的配置信息，kubelet 将会负责 Pod 的创建，同时将管理 Pod 的整个生命周期，以及向 API Server 上报 Pod 的状态。当新增节点时，kubelet 会向 API Server 注册自身节点，然后定时上报节点中资源的使用情况，并通过 cAdvisor 监控节点及节点上容器的性能。kubelet 主要功能包含以下几点：

1）节点管理。kubelet 可以向 API Server 注册自己，并定时把所在节点采集的资源信息和使用情况并提交给 API Server，当 kubelet 的启动参数设置为--register-node=true 时，kubelet 可以自动向 API Server 注册自己，这样通过启动或者停止 kubelet 进程实现节点的自动扩容、缩容。

2）Pod 管理。kubelet 获取到节点上的 Pod 清单，对比清单的 Pod 与当前节点的 Pod，如果有新增 Pod，则按照 Pod 清单的要求创建清单；如果对比发现有 Pod 修改或删除，则 kubelet 也会做出对应的操作。

Pod 清单的获取有以下三种方式：第一种通过文件获得，文件通常放在/etc/kubernetes/manifests 目录下面，启动 kubelet 时可以通过指定--config 覆盖；第二种也是通过文件获得，只不过文件是通过 URL 获取的，URL 可以在启动 kubelet 时通过--manifest-url 指定；第三种是通过 watch API Server 获取，Pod 的清单是被存在 etcd 中的。其中，前两种模式下称 kubelet 运行在 standalone 模式。

3）健康检查。创建了 Pod 之后，kubelet 要查看 Pod 中的容器是否正常运行。Kubernetes 提供了两类探针检查容器是否健康：LivenessProbe 探针和 ReadinessProbe 探针，并且 kubelet 会定时执行探针来判断容器的健康状态。

LivenessProbe 探针用于判断容器是否存活，如果探针检测到容器不处于存活状态，kubelet 根据策略重新启动容器；如果容器没有设置 LivenessProbe 探针，则 kubelet 将认为该容器始终处于存活状态。

ReadinessProbe 探针用于判断容器是否准备就绪可以接受流量访问，如果探针检测到容器无法正常访问，则 Pod 的状态将被修改，并且 Endpoint Controller 会从 Pod 对应的 Endpoint 中删除该容器的 Endpoint 记录。

对于探针检测目前提供了三种方式：HTTPGet，HTTP 的方式发送 GET 请求；tcpSocket，TCP Socket 方式访问目的端口；exec，在容器内执行命令。

4）cAdvisor 资源监控。cAdvisor（Container Advisor）是一个开源的容器资源使用监控和性能分析 agent，在 Kubernetes v1.3 中已经被 kubelet 集成，cAdvisor 可以自动发现宿主机上的容器并监控容器数据，包括 CPU、Memory、Filesystem 和 Network 等，同时对宿主机上的资源使用情况也进行了监控。在 Kubernetes 集群中，cAdvisor 默认暴露 4194 端口，并提供了简单的 Web 访问。Kubernetes 对各个节点的资源使用信息进行监控，这将帮助用户更加深入地了解应用的执行情况，并找到可能的瓶颈。

Metrics Server 项目（在 1.8 版本开始替代了 Heapster）为 Kubernetes 提供了一个基本的监控平台，Metrics Server 首先从 Kubernetes Master 获取集群中所有节点的信息，然后通过这些节点上的 kubelet 获取有用数据，而 kubelet 本身的数据则是从 cAdvisor 得到。所有获取到的数据都被推到 Metrics Server 配置的后端存储中，并还支持数据的可视化。常用后端存储和可视化的方法，如 InfluxDB + Grafana。

5. Kube-Proxy

Service 通过 Selector 选择的一组 Pods 的服务抽象，其实就是一个微服务，提供了服务的负载均衡和反向代理的能力，而 Kube-Proxy 的主要作用就是负责管理 Service 的访问入口，包括集群内 Pod 到 Service 的访问和集群外访问 Service。Kube-Proxy 运行在 Kubernetes 集群的每个节点上，负责监听 API Server 中 Service 和 Endpoint 的变化情况，并将服务的 TCP 或 UDP 流量以负载均衡的方式向 Pod 的转发。

Kube-Proxy 进行服务的负载均衡方式如图 2-9 所示。

Kube-Proxy 主要采用 Userspace、Iptables 或 IPVS 等方式实现：

- Userspace：Kubernetes 最早的方案，通过在用户空间监听一个端口，然后所有服务都通过 Iptables 转发到这个端口，再由 Kube-Proxy 负载均衡到实际的 Pod。该方式最大的问题是，Service 的请求会先从用户空间进入内核 Iptables，然后回到用户空间，由 Kube-Proxy 完成后端 Endpoint 的选择和代理工作，这样流量从用户空间进出内核带来的性能损耗是不可接受的。

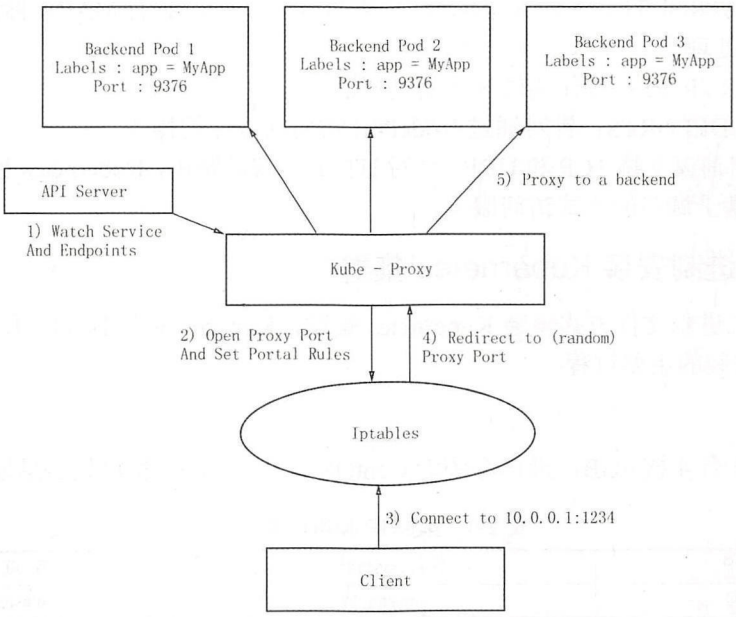


图 2-9 kube-proxy 进行服务的负载均衡方式

- Iptables：在 Kubernetes v1.2 引入，完全以 Linux 的 Iptables 规则对报文的流向进行修改来实现 Service 到 Pod 的数据转发。该方式最主要的问题是在集群规模大、服务多的时候 Iptables 规则数量十分庞大，每增加或更新一条规则存在较长的时延，大规模情况下有明显的性能问题。
- IPVS：为解决 Iptables 模式的性能问题，Kubernetes v1.8 新增了 IPVS 模式。IPVS 是 LVS 项目的一部分，基于 Netfilter 之上的协议，运行在 Linux kernel 当中的 4 层负载均衡器，性能异常优秀，有效地解决了大规模情况下的性能问题。

在上述三种方式中，最常用的是 Iptables。Iptables 规则如图 2-10 所示。

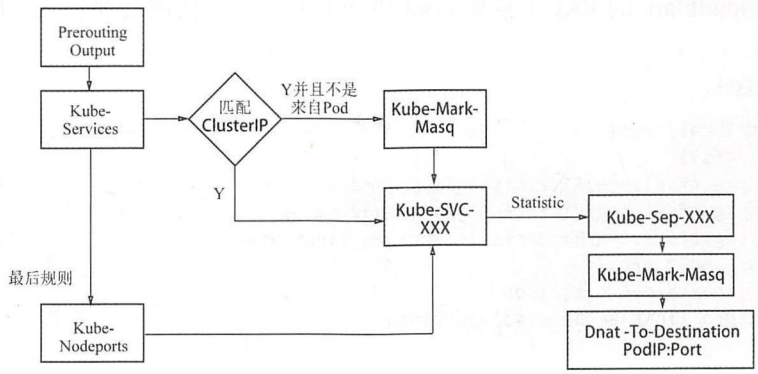


图 2-10 Iptables 规则

Kube-Proxy 对 Iptables 的链进行了扩充，自定义了 KUBE-SERVICES、KUBE-NODEPORTS、KUBE-POSTROUTING、KUBE-MARK-MASQ 和 KUBE-MARK-DROP 五个链。

- KUBE-MARK-MASQ：对于符合条件的包设置 mark 0x4000，有此标记的数据包会在 KUBE-POSTROUTING 链中统一做 MASQUERADE。
- KUBE-MARK-DROP：对于未能匹配到跳转规则的设置 mark 0x8000，有此标记的数据包会在 filter 表中丢掉。
- KUBE-POSTROUTING：对 NODE 节点上匹配 Kubernetes 独有 MARK 标记的数据包，进行 SNAT 处理。
- KUBE-SERVICES：操作跳转规则的主要链。
- KUBE-NODEPORTS：针对通过 NodePort 访问的包做的操作。

Kube-Proxy 目前仅支持 TCP 和 UDP。针对 HTTP 协议的路由，Kubernetes 也提供了 Ingress Controller，通过基于域名的方式访问服务。

2.2.3 二进制安装 Kubernetes 集群

本节介绍用二进制文件方式安装 Kubernetes 集群，Kubernetes 选择 v1.10，集群网络采用 flannel。以下为安装的主要过程。

1. 环境准备

集群机器为 4 台 4 核 4GB，操作系统为 CentOS 7.3。集群 IP 和角色信息如表 2-1 所示。

表 2-1 集群 IP 和角色信息

节 点 IP	节点 hostname	节 点 角 色
192.16.1.64	master-1	master+etcd
192.16.1.65	master-2	master+etcd
192.16.1.66	master-3	master+etcd
192.16.1.67	node-1	node

编辑/etc/hosts 文件，配置 hostname 通信。

```
vi /etc/hosts
192.16.1.64  master-1
192.16.1.65  master-2
192.16.1.66  master-3
192.16.1.67  node-1
```

2. 创建证书

本书使用 CloudFlare 的 PKI 工具集 cfssl 生成 CA 证书、秘钥文件和各个组件所需要的证书。

首先安装 cfssl。

```
mkdir -p /opt/local/cfssl
cd /opt/local/cfssl
wget https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
wget https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
wget https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64
mv cfssl_linux-amd64 cfssl
mv cfssljson_linux-amd64 cfssljson
mv cfssl-certinfo_linux-amd64 cfssl-certinfo
chmod +x *
```

然后配置 CA 证书。

```
mkdir /opt/ssl
cd /opt/ssl

# 配置 config.json 及 csr.json 文件
vi  config.json
vi csr.json
```

之后生成 CA 证书并分发。

```
cd /opt/ssl/
/opt/local/cfssl/cfssl gencert -initca csr.json |  \ /opt/local/cfssl/cfssljson -bare ca

# 创建证书目录
mkdir -p /etc/kubernetes/ssl
cp *.pem /etc/kubernetes/ssl

# 这里要将文件复制到集群所有机器上，以 master-2 为例
scp *.pem 192.16.1.65:/etc/kubernetes/ssl/
```

最后生成 admin、kubernetes、etcd、kube-proxy 证书。

```
# 生成 admin 证书和私钥
vi admin-csr.json
/opt/local/cfssl/cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem \
  -ca-key=/etc/kubernetes/ssl/ca-key.pem \
  -config=/opt/ssl/config.json \
  -profile=kubernetes admin-csr.json | /opt/local/cfssl/cfssljson -bare admin

# 生成 Kubernetes 证书和私钥
vi kubernetes-csr.json
/opt/local/cfssl/cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem \
  -ca-key=/etc/kubernetes/ssl/ca-key.pem \
  -config=/opt/ssl/config.json \
  -profile=kubernetes kubernetes-csr.json | /opt/local/cfssl/cfssljson -bare kubernetes

# 生成 etcd 证书和私钥
vi etcd-csr.json
/opt/local/cfssl/cfssl gencert -ca=/opt/ssl/ca.pem \
  -ca-key=/opt/ssl/ca-key.pem \
  -config=/opt/ssl/config.json \
  -profile=kubernetes etcd-csr.json | /opt/local/cfssl/cfssljson -bare etcd

# 这里要将 admin、Kubernetes、etcd 证书文件拷贝到集群所有 master 机器上，以 master-2 为例
cp admin*.pem /etc/kubernetes/ssl/
cp kubernetes *.pem /etc/kubernetes/ssl/
cp etcd*.pem /etc/kubernetes/ssl/
scp admin*.pem 172.16.1.65:/etc/kubernetes/ssl/
scp kubernetes*.pem 172.16.1.65:/etc/kubernetes/ssl/
scp etcd*.pem 172.16.1.65:/etc/kubernetes/ssl/

# 生成 kube-proxy 证书和私钥，通过 master 节点生成下发到 node 节点上
vi kube-proxy-csr.json
/opt/local/cfssl/cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem \
  -ca-key=/etc/kubernetes/ssl/ca-key.pem \
  -config=/opt/ssl/config.json \
  -profile=kubernetes  kube-proxy-csr.json | /opt/local/cfssl/cfssljson -bare kube-proxy

# 这里要将 kube-proxy 证书文件拷贝到集群所有 node 机器上
cp kube-proxy*.pem /etc/kubernetes/ssl/
scp kube-proxy*.pem 192.16.1.67:/etc/kubernetes/ssl/
```

3. 安装 Docker

```
# 安装指定版本 docker-ce 17.03 依赖 docker-ce-selinux，不能直接用 yum 安装 docker-ce
wget \ https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-ce-selinux-17.03.1.ce-1.el7.centos.noarch.rpm
```

```
rpm -ivh docker-ce-selinux-17.03.1.ce-1.el7.centos.noarch.rpm
yum -y install docker-ce-17.03.1.ce
# 配置使用 overlay2（使用 overlay2 建议将系统内核更新到 kernel 4.0）
vi /etc/docker/daemon.json
{
  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ]
}
# 配置 docker dns，10.254.0.2 为 coredns 的 clusterIp
vi /etc/systemd/system/docker.service.d/docker-dns.conf
[Service]
Environment="DOCKER_DNS_OPTIONS=\
    --dns 10.254.0.2 --dns 114.114.114.114  \
    --dns-search default.svc.cluster.local --dns-search svc.cluster.local  \
    --dns-opt ndots:2 --dns-opt timeout:2 --dns-opt attempts:2"
```

4. 安装 etcd 集群

下载版本为 3.1.12 的 etcd。

```
wget https://github.com/coreos/etcd/releases/download/v3.1.12/etcd-v3.1.12- linux-amd64.tar.gz
tar zxvf etcd-v3.1.11-linux-amd64.tar.gz
cd etcd-v3.1.11-linux-amd64
mv etcd  etcdctl /usr/bin/

# 配置启动 etcd，也可以做成服务方式启动
etcd \
  --name=etcd1 \
  --cert-file=/etc/kubernetes/ssl/etcd.pem \
  --key-file=/etc/kubernetes/ssl/etcd-key.pem \
  --peer-cert-file=/etc/kubernetes/ssl/etcd.pem \
  --peer-key-file=/etc/kubernetes/ssl/etcd-key.pem \
  --trusted-ca-file=/etc/kubernetes/ssl/ca.pem \
  --peer-trusted-ca-file=/etc/kubernetes/ssl/ca.pem \
  --initial-advertise-peer-urls=https://192.16.1.64:2380 \
  --listen-peer-urls=https://192.16.1.64:2380 \
  --listen-client-urls=https://192.16.1.64:2379,http://127.0.0.1:2379 \
  --advertise-client-urls=https://192.16.1.64:2379 \
  --initial-cluster-token=k8s-etcd-cluster \
  --initial-cluster=etcd1=https://192.16.1.64:2380,etcd2=https://192.16. 1.65:2380,etcd3=https:
//192.16.1.66:2380 \
  --initial-cluster-state=new \
  --data-dir=/opt/etcd/
```

5. 安装 Kubernetes 组件

在 master 节点上安装 kube-apiserver、kube-scheduler、kube-controller-manager 三个组件以及客户端 kubectl，在 node 上安装 kube-proxy 和 kubelet 组件。Flannel 组件在 master 和 node 上都要安装。

```
# master 节点
cd /tmp
wget https://dl.k8s.io/v1.10.0/kubernetes-server-linux-amd64.tar.gz
wget \ https://github.com/coreos/flannel/releases/download/v0.9.0/flannel-v0.9.0-linux-amd64.
tar.gz
tar -xzvf kubernetes-server-linux-amd64.tar.gz
tar -xzvf flannel-v0.9.0-linux-amd64.tar.gz
cp  -r  kubernetes/server/bin/{kube-apiserver,kube-controller-manager,kube-scheduler,kubectl}
/usr/local/bin/
cp -r flanneld /usr/local/bin/

# node 节点
cd /tmp
```

```
    wget https://dl.k8s.io/v1.10.0/kubernetes-server-linux-amd64.tar.gz
    wget \ https://github.com/coreos/flannel/releases/download/v0.9.0/flannel-v0.9.0-linux-amd64.
tar.gz
    tar -xzvf kubernetes-server-linux-amd64.tar.gz
    tar -xzvf flannel-v0.9.0-linux-amd64.tar.gz
    cp -r kubernetes/server/bin/{kube-proxy,kubelet} /usr/local/bin/
    cp -r flanneld /usr/local/bin/
```

6. Bootstrap Token

由于通过手动创建 CA 方式太过烦琐，只适合少量机器，因为每次签证时都需要绑定 Node IP，随机器增加会带来很多困扰。因此，本书使用 TLS Bootstrapping 方式进行授权，由 APIServer 自动给符合条件的 node 发送证书来授权加入集群。

kubelet 第一次启动时会向 kube-apiserver 发送 TLS Bootstrapping 请求，kube-apiserver 验证 kubelet 请求中的 Token 是否与它配置的 Token 一致，如果一致则自动为 kubelet 生成证书和秘钥。

```
    # 生成 Token 及创建 token.csv 文件
    export BOOTSTRAP_TOKEN=$(head -c 16 /dev/urandom | od -An -t x | tr -d ' ')
    cat > /etc/kubernetes/token.csv <<EOF
    ${BOOTSTRAP_TOKEN},kubelet-bootstrap,10001,"system:kubelet-bootstrap"
    EOF

    scp /etc/kubernetes/token.csv 192.16.1.65:/etc/kubernetes/
    scp /etc/kubernetes/token.csv 192.16.1.66:/etc/kubernetes/

    kubectl create clusterrolebinding kubelet-bootstrap --clusterrole=system:node-bootstrapper
--user=kubelet-bootstrap

    # 创建 bootstrap kubeconfig 文件
    kubectl config set-cluster kubernetes \
      --certificate-authority=/etc/kubernetes/ssl/ca.pem \
      --embed-certs=true \
      --server=https://127.0.0.1:6443 \
      --kubeconfig=bootstrap.kubeconfig

    kubectl config set-credentials kubelet-bootstrap \
      --token=$(BOOTSTRAP_TOKEN) \
      --kubeconfig=bootstrap.kubeconfig

    kubectl config set-context default \
      --cluster=kubernetes \
      --user=kubelet-bootstrap \
      --kubeconfig=bootstrap.kubeconfig

    kubectl config use-context default --kubeconfig=bootstrap.kubeconfig
```

7. 配置启动 kube-apiserver

```
    cd /etc/kubernetes
    cat >> audit-policy.yaml <<EOF
    # Log all requests at the Metadata level.
    apiVersion: audit.k8s.io/v1beta1
    kind: Policy
    rules:
    - level: Metadata
    EOF

    # 启动 kube-apiserver，也可以做成服务方式启动
    kube-apiserver \
      --admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,
ResourceQuota,NodeRestriction \
      --advertise-address=192.16.1.64 \
```

```
  --allow-privileged=true \
  --apiserver-count=3 \
  --audit-policy-file=/etc/kubernetes/audit-policy.yaml \
  --audit-log-maxage=30 \
  --audit-log-maxbackup=3 \
  --audit-log-maxsize=100 \
  --audit-log-path=/var/log/kubernetes/audit.log \
  --authorization-mode=Node,RBAC \
  --bind-address=0.0.0.0 \
  --secure-port=6443 \
  --client-ca-file=/etc/kubernetes/ssl/ca.pem \
  --enable-swagger-ui=true \
  --etcd-cafile=/etc/kubernetes/ssl/ca.pem \
  --etcd-certfile=/etc/kubernetes/ssl/etcd.pem \
  --etcd-keyfile=/etc/kubernetes/ssl/etcd-key.pem \
  --etcd-servers=https://192.16.1.64:2379,https://192.16.1.65:2379,https://192.16.1.66:2379 \
  --event-ttl=1h \
  --kubelet-https=true \
  --insecure-bind-address=127.0.0.1 \
  --insecure-port=8080 \
  --service-account-key-file=/etc/kubernetes/ssl/ca-key.pem \
  --service-cluster-ip-range=10.254.0.0/18 \
  --service-node-port-range=30000-32000 \
  --tls-cert-file=/etc/kubernetes/ssl/kubernetes.pem \
  --tls-private-key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
  --enable-bootstrap-token-auth \
  --token-auth-file=/etc/kubernetes/token.csv \
  --v=1
```

8. 配置启动 kube-controller-manager

```
# 启动 kube-controller-manager，也可以做成服务方式启动
kube-controller-manager \
  --address=0.0.0.0 \
  --master=http://127.0.0.1:8080 \
  --allocate-node-cidrs=true \
  --service-cluster-ip-range=10.254.0.0/18 \
  --cluster-cidr=10.254.64.0/18 \
  --cluster-name=kubernetes \
  --service-account-private-key-file=/etc/kubernetes/ssl/ca-key.pem \
  --root-ca-file=/etc/kubernetes/ssl/ca.pem \
  --leader-elect=true \
  --v=1
```

9. 配置启动 kube-scheduler

```
# 启动 kube-scheduler，也可以做成服务方式启动
kube-scheduler \
  --address=0.0.0.0 \
  --master=http://127.0.0.1:8080 \
  --leader-elect=true \
  --v=1
```

10. 配置启动 kubelet

```
# 启动 kubelet，也可以做成服务方式启动
kubelet \
  --cgroup-driver=cgroupfs \
  --hostname-override=node-1 \
  --pod-infra-container-image=jicki/pause-amd64:3.0 \
  --experimental-bootstrap-kubeconfig=/etc/kubernetes/bootstrap.kubeconfig \
  --kubeconfig=/etc/kubernetes/kubelet.kubeconfig \
  --cert-dir=/etc/kubernetes/ssl \
  --cluster_dns=10.254.0.2 \
  --cluster_domain=cluster.local. \
  --hairpin-mode promiscuous-bridge \
```

```
    --allow-privileged=true \
    --fail-swap-on=false \
    --serialize-image-pulls=false \
    --logtostderr=true \
    --max-pods=512 \
    --v=1
```

11. 配置启动 kube-proxy

```
# 创建 kube-proxy kubeconfig 文件
kubectl config set-cluster kubernetes \
  --certificate-authority=/etc/kubernetes/ssl/ca.pem \
  --embed-certs=true \
  --server=https://127.0.0.1:6443 \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config set-credentials kube-proxy \
  --client-certificate=/etc/kubernetes/ssl/kube-proxy.pem \
  --client-key=/etc/kubernetes/ssl/kube-proxy-key.pem \
  --embed-certs=true \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config set-context default \
  --cluster=kubernetes \
  --user=kube-proxy \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig

scp kube-proxy.kubeconfig 192.16.1.67:/etc/kubernetes/

# 启动 kube-proxy，也可以做成服务方式启动
kube-proxy \
  --bind-address=192.16.1.67 \
  --hostname-override=node-1 \
  --cluster-cidr=10.254.64.0/18 \
  --masquerade-all \
  --proxy-mode=ipvs \
  --ipvs-min-sync-period=5s \
  --ipvs-sync-period=5s \
  --ipvs-scheduler=rr \
  --kubeconfig=/etc/kubernetes/kube-proxy.kubeconfig \
  --logtostderr=true \
  --v=1
```

12. 配置启动 flannel

```
# 配置 flannel 网段
etcdctl
--endpoints=https://192.16.1.64:2379,https://192.16.1.65:2379,https://192.16.1.66:2379 \
  --cert-file=/etc/kubernetes/ssl/etcd.pem \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --key-file=/etc/kubernetes/ssl/etcd-key.pem \
  set /flannel/network/config \ '{"Network":"10.254.64.0/18","SubnetLen":24,"Backend":{"Type":
"host-gw"}}'

# 启动 flannel，也可以做成服务方式启动
flanneld \
--etcd-cafile=/etc/kubernetes/ssl/ca.pem \
--etcd-certfile=/etc/kubernetes/ssl/etcd.pem \
--etcd-keyfile=/etc/kubernetes/ssl/etcd-key.pem \
--etcd-endpoints=https://192.16.1.64:2379,https://192.16.1.65:2379,https://192.16.1.66:2379 \
--etcd-prefix=/kubernetes/network \
--iface=eth0
```

13. 配置 CoreDns

```
# 下载 CoreDns yaml 文件
wget https://raw.githubusercontent.com/coredns/deployment/master/kubernetes/coredns.yaml.sed
mv coredns.yaml.sed coredns.yaml

# vi coredns.yaml
# kubernetes cluster.local 为创建 svc 的 IP 段，clusterIP 为指定 DNS 的 IP
kubernetes cluster.local 10.254.0.0/18
clusterIP: 10.254.0.2

# 创建 coreDns
kubectl apply -f coredns.yaml
```

14. 配置 kubectl

生成集群管理员 admin 的 kubeconfig 文件给 kubectl 调用。

```
# 配置 kubectl kubeconfig 文件
kubectl config set-cluster kubernetes \
  --certificate-authority=/etc/kubernetes/ssl/ca.pem \
  --embed-certs=true \
  --server=https://127.0.0.1:6443

kubectl config set-credentials admin \
  --client-certificate=/etc/kubernetes/ssl/admin.pem \
  --embed-certs=true \
  --client-key=/etc/kubernetes/ssl/admin-key.pem

kubectl config set-context kubernetes \
  --cluster=kubernetes \
  --user=admin

kubectl config use-context kubernetes

cp /root/.kube/config /etc/kubernetes/kubelet.kubeconfig
scp /etc/kubernetes/kubelet.kubeconfig 192.16.1.65:/etc/kubernetes/
scp /etc/kubernetes/kubelet.kubeconfig 192.16.1.66:/etc/kubernetes/
```

2.2.4 kubespray 安装 Kubernetes 集群

kubespray 是 Kubernetes incubator 中的项目，目标是提供 Kubernetes 生产级的快速部署方案，kubespray 的基础是通过 Ansible Playbook 定义系统与 Kubernetes 集群部署的任务，具有以下几个特点：

- 可以部署在 AWS、GCE、Azure、OpenStack 和裸机上。
- 部署高可用 Kubernetes 集群。
- 可组合性，可自行选择网络插件（flannel、calico、canal、weave）部署。
- 支持多种 Linux distributions（CoreOS、Debian Jessie、Ubuntu 16.04、CentOS/RHEL7）。

以下为 kubespray 安装的主要过程，集群环境与上一节保持一致，同时增加一个 ansible server 角色机器，kubespray 选择 v2.4.0 版本。

1. 环境准备

集群机器为 5 台 4 核 4G，操作系统为 CentOS 7.3。集群 IP 和角色信息如表 2-2 所示。

表 2-2 集群 IP 和角色信息

节 点 IP	节点 hostname	节 点 角 色
192.16.1.63	ansible	ansible-server

续表

节 点 IP	节点 hostname	节 点 角 色
192.16.1.64	master-1	master+etcd
192.16.1.65	master-2	master+etcd
192.16.1.66	master-3	master+etcd
192.16.1.67	node-1	node

ansible-server 需要安装 ansible v2.4 及以上版本、Jinja 2.9 及以上版本和 python-netaddr。

```
# 在 ansible-server 上进行操作
yum install -y python-pip python-netaddr ansible git
pip install --upgrade Jinja2
```

配置 SSH Key 登入。

```
ssh-keygen -t rsa -N ""

ssh-copy-id -i /root/.ssh/id_rsa.pub 192.16.1.64
ssh-copy-id -i /root/.ssh/id_rsa.pub 192.16.1.65
ssh-copy-id -i /root/.ssh/id_rsa.pub 192.16.1.66
ssh-copy-id -i /root/.ssh/id_rsa.pub 192.16.1.67
```

2. 获取 kubespray

```
git clone https://github.com/kubernetes-incubator/kubespray.git
cd kubespray
git checkout v2.4.0 -b myv2.4.0
cp inventory/inventory.example inventory/inventory.cfg
```

3. 配置 inventory

ansible 中 inventory.cfg 文件指定需要操作的主机列表，以及各个主机的角色。

```
vi inventory/inventory.cfg
[all]
master-1 ansible_ssh_host=192.16.1.64 ip=192.16.1.64
master-2 ansible_ssh_host=192.16.1.65 ip=192.16.1.65
master-3 ansible_ssh_host=192.16.1.66 ip=192.16.1.66
node-1 ansible_ssh_host=192.16.1.67 ip=192.16.1.67
[kube-master]
master-1
master-2
master-3
[etcd]
master-1
master-2
master-3
[kube-node]
node-1
[k8s-cluster:children]
kube-node
kube-master
```

4. Kubernetes 集群配置

```
cp inventory/group_vars/all.yml inventory/group_vars/all.yml.bak
cp inventory/group_vars/k8s-cluster.yml inventory/group_vars/k8s-cluster.yml.bak

vi inventory/group_vars/all.yml
# 指定操作系统为 centos
bootstrap_os: centos

vi inventory/group_vars/k8s-cluster.yml
# 指定网络方案为 flannel
```

```
kube_network_plugin: flannel
# 默认关闭 dashboard
dashboard_enabled: false
```

5. 镜像源修改

修改下载的包，将国外下载慢的包改成用阿里的源。

```
vi roles/download/defaults/main.yml
etcd_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/etcd"
flannel_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/flannel"
flannel_cni_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/flannel-cni"
hyperkube_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/hyper-kube"
pod_infra_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/pause-amd64"
nginx_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/nginx"
kubedns_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/k8s-dns-kube-dns-amd64"
dnsmasq_nanny_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/k8s-dns-dnsmasq-nanny-
amd64"
dnsmasq_sidecar_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/k8s-dns-sidecar-amd64"
kubednsautoscaler_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/cluster-proportional-
autoscaler-amd64"

vi roles/kubernetes-apps/ansible/defaults/main.yml
kubedns_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/k8s-dns-kube-dns-amd64"
dnsmasq_nanny_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/k8s-dns-dnsmasq-nanny-
amd64"
dnsmasq_sidecar_image_repo: "registry.cn-hangzhou.aliyuncs.com/linkcloud/k8s-dns-sidecar-amd64"
kubednsautoscaler_image_repo:  "registry.cn-hangzhou.aliyuncs.com/linkcloud/cluster-proportional-
autoscaler-amd64"
```

6. ansible 进行集群部署

```
ansible-playbook -b -i inventory/inventory.cfg cluster.yml --flush-cache
```

2.3 Kubernetes 实战案例

在 Kubernetes 下，应用能很好地完成发布、升级、扩缩容等动作，特别是 Kubernetes 对微服务有很好的支持，应用本身通过 Deployment 进行部署，各个服务运行在 Pod 中，Pod 之间的服务通过 Service 具备的服务发现实现相互间的通信。本部分将以一个搭建 WordPress 博客应用的案例讲解如何快速实践 Kubernetes。

2.3.1 WordPress 应用模型

WordPress 架构非常简洁，由 WordPress Server 对外提供 Web 服务，以及 MariaDB 作为 WordPress Server 的后端存储，然后即可通过浏览器进行访问。WordPress 应用模型如图 2-11 所示。

图 2-11 WordPress 应用模型

WordPress 应用分为 WordPress Server 和 MariaDB 两个 Deployment 进行部署，然后相互之间通过 Service 进行访问。

2.3.2 部署 WordPress

WordPress Server 部署需要创建的对象如下。

- wordpress_deployment：Deployment 对象用于部署 WordPress。
- wordpress_service：Service 对象用于提供 WordPress 服务对外的访问。

1. WordPress Deployment

WordPress Deployment yaml 的定义如下。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - name: wordpress
        image: wordpress:4.8.1
        imagePullPolicy: IfNotPresent
        env:
        - name: ALLOW_EMPTY_PASSWORD
          value: "yes"
        - name: MARIADB_HOST
          value: mariadb_service
        - name: MARIADB_PORT_NUMBER
          value: "3306"
        - name: WORDPRESS_DATABASE_NAME
          value: wordpress
        - name: WORDPRESS_DATABASE_USER
          value: wordpress
        - name: WORDPRESS_DATABASE_PASSWORD
          value: 123456
        - name: WORDPRESS_USERNAME
          value: user
        - name: WORDPRESS_PASSWORD
value: 123456
        ports:
        - name: http
          containerPort: 80
        - name: https
          containerPort: 443
```

上述 yaml 中定义了名为 WordPress 的 Deployment，spec.replicas=1 指定了 WordPress Pod 副本个数为 1 个，当副本数小于 1 时，ReplicaSet 会生成新的 Pod。上述 Pod 中只包含一个 WordPress 的容器，指定了容器 HTTP、HTTPS 两个 port，同时指定了容器中的环境变量，这些环境变量将成为容器运行中需要用的信息，包括 MariaDB 地址、MariaDB 端口、MariaDB 用户、MariaDB 密码，以及 WordPress 用户密码等信息。在实际环境中，可以使用 Kubernetes Secret 对象存储 MariaDB 密码、WordPress 密码等敏感信息。

执行以下命令进行创建 Deployment。

```
$ kubectl create -f wordpress_deployment.yaml
deployment "wordpress" created
```

查看 ReplicaSet 创建的情况。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
wordpress-2989759959           1         1         1         1m
```

可以看到生成了一个 wordpress–xxxxx 的 ReplicaSet 对象，DESIRED、CURRENT、READY 值都为 1，表明了 wordpress Pod 数量期望为 1 个，现在已经存在 1 个，并且处于 Ready 状态。查看 Pod 创建的情况。

```
$ kubectl get pod
NAME                                READY     STATUS    RESTARTS   AGE
wordpress-2989759959-vrbk2          1/1       Running   0          1m
```

这里看到有一个 wordpress -xxxxx-yyyyy 的 Pod，其中 xxxxx 与上面 ReplicaSet 中的一致。这里刚开始 Pod 为 Pending 状态，因为下载镜像及启动需要一定时间，当 Pod 启动完成后 STATUS 将更新成 Running 状态。

如果想对 Deployment 的 Pod 数进行扩容缩容，例如对 WordPress 扩容成 10 个 Pod。

```
$ kubectl scale deployment wordpress --replicas 10
deployment "wordpress " scaled
```

2. WordPress Service

WordPress Service yaml 定义如下。

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  type: NodePort
  ports:
    - name: http
      port: 80
      targetPort: http
    - name: https
      port: 443
      targetPort: https
  selector:
    app: wordpress
```

上述 yaml 中定义了名为 wordpress 的 Service，Service 通过 selector app:wordpress 与上一小节创建的 Pod 进行绑定。这里 spec.type=NodePort 表明提供主机端口映射成容器端口的方式进行外网访问，其中的 targetPort 则是上一小节中容器中对应的 port 名称。

执行以下命令进行创建 Service。

```
$ kubectl create -f wordpress_service.yaml
service "wordpress" created
```

查看 Service 创建的情况。

```
NAME        CLUSTER-IP     EXTERNAL-IP    PORT(S)                      AGE
wordpress   10.233.22.245   <nodes>       80:32644/TCP,443:30352/TCP   1m
```

可以看出 WordPress Service 已经创建成功，80 端口映射到了 node 的 32644 端口，443 端口映射到了 node 的 30352 端口。

2.3.3 部署 MariaDB

MariaDB 部署需要创建的对象如下：

- mariadb_deployment：Deployment 对象，用于部署 MariaDB。

- mariadb_service：Service 对象，用于提供 MariaDB 服务对外的访问。
- mariadb_pvc：PersistentVolumeClaim 对象，用于对 MariaDB 进行卷的持久化声明。

1. MariaDB Deployment

MariaDB Deployment yaml 定义如下。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mariadb
  labels:
    app: mariadb
spec:
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
      - name: mariadb
        image: mariadb.10.1.23
        imagePullPolicy: IfNotPresent
        env:
        - name: MARIADB_ROOT_PASSWORD
          key: mariadb-root-password
        - name: MARIADB_PASSWORD
          key: 123456
        - name: MARIADB_USER
          value: wordpress
        - name: MARIADB_DATABASE
          value: wordpress
        ports:
        - name: mariadb
          containerPort: 3306
        volumeMounts:
        - name: data
          mountPath: /mariadb
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: mariadb
```

上述 yaml 定义了名为 mariadb 的 Deployment，spec.replicas=1 指定了 MariaDB Pod 副本个数为 1 个，并指定了容器中的环境变量，这些环境变量信息包括 MariaDB 数据库名、MariaDB 用户、MariaDB 密码。在实际环境中可以使用 Kubernetes Secret 对象存储 MariaDB 密码信息。另外，在 MariaDB 容器中指定了一个 data 的卷并挂载在/mariadb 目录下。

执行以下命令，创建 Deployment。

```
$ kubectl create -f mariadb_deployment.yaml
deployment "mariadb" created
```

查看 ReplicaSet 创建的情况。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
mariadb-3443813747             1         1         1         1m
wordpress-2989759959           1         1         1         10m
```

查看 Pod 创建的情况。

```
$ kubectl get pod
NAME                             READY     STATUS    RESTARTS   AGE
mariadb-3443813747-43zh6         1/1       Running   0          1m
wordpress-2989759959-vrbk2       1/1       Running   0          10m
```

2. MariaDB Service

MariaDB Service yaml 定义如下。

```
apiVersion: v1
kind: Service
metadata:
  name: mariadb_service
  labels:
    app: mariadb
spec:
  ports:
    - name: mariadb
      port: 3306
      targetPort: mariadb
  selector:
    app: mariadb
```

上述 yaml 中定义了名为 mariadb 的 Service，Service 通过 selector app:mariadb 与前文创建的 Pod 进行绑定。由于没有 spec.type 字段，默认为 ClusterIp 只允许 Kubernetes 内部 Pod 之间访问。

执行以下命令，创建 Service。

```
$ kubectl create -f mariadb_service.yaml
service "mariadb" created
```

查看 Service 创建的情况。

```
$ kubectl get rs
NAME       CLUSTER-IP    EXTERNAL-IP    PORT(S)                    AGE
mariadb     10.233.54.8     <none>      3306/TCP                   1m
wordpress   10.233.22.245   <nodes>    80:32644/TCP,443:30352/TCP  10m
```

3. MariaDB PersistentVolumeClaim

MariaDB PersistentVolumeClaim yaml 定义如下。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mariadb
  labels:
    app: mariadb
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: "kubernetes.io/rbd"
```

上述 yaml 中定义了名为 mariadb 的 PersistentVolumeClaim，spec.accessModes=["ReadWriteOnce"] 表明 PVC 以单个 Pod 以读写模式挂载，storage=10Gi 定义了 PVC 声明了 10GB 大小，storageClassName=kubernetes.io/rbd 指明了以 kubernetes.io/rbd 提供方提供 PersistentVolume 资源。

执行以下命令，创建 PersistentVolumeClaim。

```
$ kubectl create -f mariadb_pvc.yaml
persistentVolumeClaim "mariadb" created
```

查看 PersistentVolume 创建的情况。

```
$ kubectl get pv
NAME                                       CAPACITY        ACCESSMODES
```

```
RECLAIMPOLICY   STATUS        CLAIM          STORAGECLASS    REASON    AGE
pvc-2ff77d2a-8974-11e7-9391-d4bed9ef0f0b        10Gi            RWO
Delete          Bound    default/mariadb  kubernetes.io/rbd          1m
```

上面可以看出，PVC 创建的同时依赖于 storageClassName=kubernetes.io/rbd 创建了 PV 资源，BOUND 状态表示创建的 PV 资源已经被挂载到容器上了。

2.3.4　通过浏览器访问 WordPress

上述的几个小节完成了 WordPress 在 Kubernetes 中所需要的各类资源的创建。下面通过浏览器进行访问。由于 WordPress Service 资源是通过 NodePort 方式进行服务的暴露，而对应的 80 端口为 32644，对应的 443 端口为 30352，因此可以通过 http://node ip:32644 访问 WordPress，以及通过 https://node ip:30352 以 HTTPS 的方式访问 WordPress，访问如图 2-12 所示。表明 WordPress 已经部署成功，接下来可以使用 WordPress 作为自己的博客。

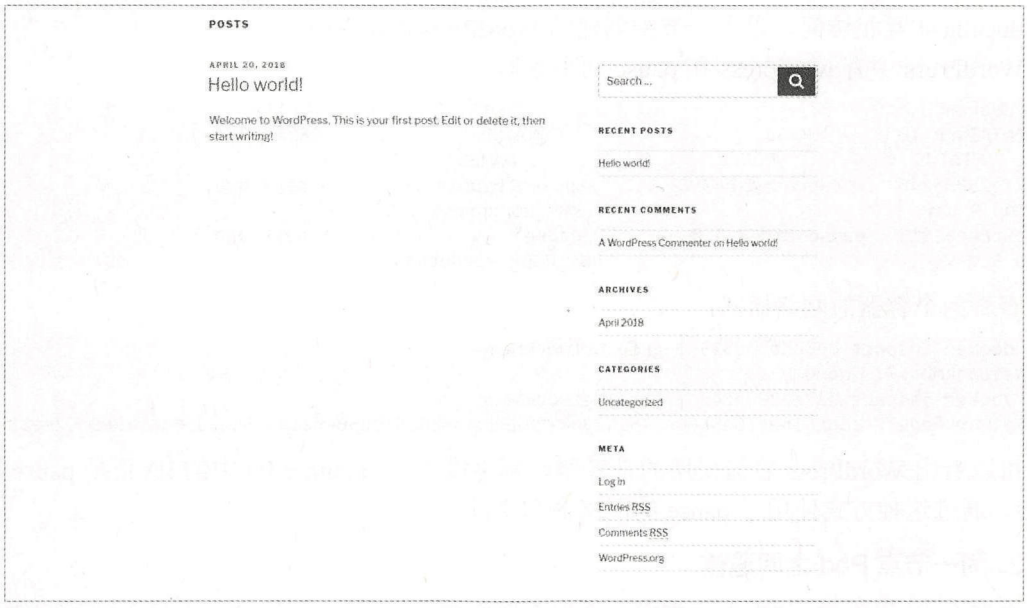


图 2-12　访问 WordPress

到此，已经完成了 WordPress 在 Kubernetes 上的实践案例。本例是一个典型的 Web 应用，相对比较简单，但能很好地指导学习 Kubernetes，并从这个例子出发完成对复杂架构应用的部署。

2.4　Kubernetes 网络

Kubernetes 是一个跨主机的容器编排管理平台，运行在 Kubernetes 上的应用是如何被访问的呢？实现应用访问的网络则正是 Kubernetes 中最核心的部分之一。本部分将从网络场景、网络模型和开源网络方案三个维度讲解 Kubernetes 网络。

2.4.1　Kubernetes 中的网络场景

Kubernetes 默认的网络方式与 Docker 有所不同，需要解决四种网络接入场景。

- Pod 中容器与容器之间的网络通信。
- Pod 与 Pod 之间的网络通信。

- Pod 与 Service 之间的网络通信。
- Kubernetes 集群外部与 Service 之间的网络通信。

其中，Pod 与 Service 之间的通信以及集群外部与 Service 之间通信核心依赖于 Kube-Proxy 组件，网络场景已经在之前场景介绍过。本节将对前两个场景进行讲解，其中 Pod 与 Pod 之间的网络通信为 Kubernetes 网络场景的重点，而针对 Pod 之间的通信又分为两种情况：Pod 在同一个节点上进行通信、Pod 在不同的节点上进行通信。下面针对这几种情况分别进行分析。

1. Pod 中容器之间的网络通信

在 Kubernetes 中，Pod 是最小的操作单元，在 Pod 中可以运行多个容器。在第 2.1.4 节中提到 Pod 中运行着一个 Pause 容器，为 Pod 提供网络命名空间，Pod 中所有容器都共享着这个网络空间。在实现上，如 Docker 提供了--net=container:ID 这样的选项。每个 Pod 对外呈现一个唯一的 IP，Pod 内的各容器共享这个 IP，也就是说各容器处于同一个网络空间中，可直接使用 localhost 互相访问。以上一节中创建的 WordPress Pod 为例。

WordPress 中有 wordpress 和 pause 两个容器：

```
$ docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED
    STATUS              PORTS                  NAMES
a27e2309a246   wordpress@sha256:...  "/app-entrypoint.s..."  5 days ago
  Up 5 days                             k8s_wordpress-...
0e5c6bebe515   pause-amd64:3.0        "/pause"                5 days ago
Up 5 days                            k8s_POD_ wordpress-...
```

查看两个容器的网络模式：

```
$ docker inspect 0e5c6bebe515 | grep NetworkMode
"NetworkMode": "none",
$ docker inspect a27e2309a246 | grep NetworkMode
"NetworkMode":"container:0e5c6bebe515236072833dd48488151b8564001d2359ce3f04a519e9f02ee327",
```

可以看出 WordPress 容器采用的是容器间网络模式，container:ID 中的 ID 正是 pause 容器的 ID，通过这种方式使用了 pause 的网络命名空间。

2. 同一节点 Pod 之间通信

在每个 Kubernetes 节点上，都有一个根命名空间（root netns）。节点上最主要的网络接口 eth0 是在这个 root netns 下。同时，每个 Pod 也有其自身的 netns，通过 veth pair 虚拟网卡对连接到 root netns，其中一端在 root netns 内，另一端在 Pod 的 nens 内。通常在 Pod 端的网络接口为 eth0，这样在 Pod 中不需要感知底层主机的网络结构，另一端命名成比如 vethxxx。这些 Pod 要相互通信，就需要通过网桥的中转，Docker 提供了 docker0 网桥。图 2-13 所示为同一节点中 Pod 通信的网络结构。

具体 Pod1 数据包如何被 Pod2 接收呢？首先数据包从 Pod1 中 netns 的 eth0 发出，通过 vethxxx 进入 root netns；然后被传到 docker0 网桥，通过 ARP 协议，发现 docker0 转发的目标设备 vethyyy；数据包到达 vethyyy，通过 veth pair 到达 Pod2 中 netns 的 eth0。

3. 不同节点 Pod 之间通信

上节中对于 Pod 在同一个节点，利用 docker0 网桥进行了中转。如果两个 Pod 在不同节点上时，则需要解决两个问题：由于 docker0 网桥无法实现跨主机的中转，因此需要解决 Pod 跨主机通信问题；不同节点上 docker 的网段可能会出现重复的情况，因此需要解决 Pod 间 IP 冲突的问题。

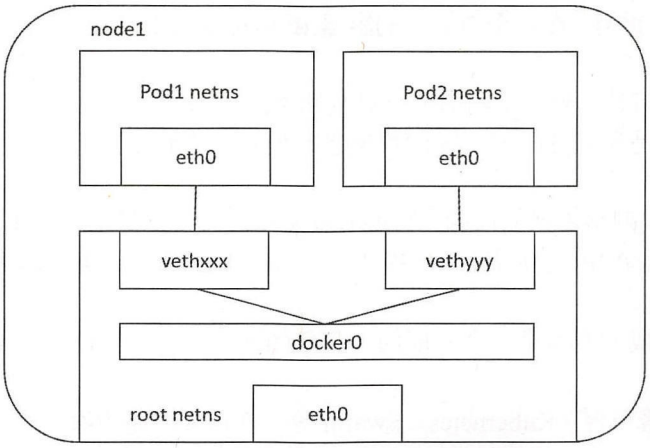


图 2-13　同一节点中 Pod 通信的网络结构

图 2-14 所示为不同节点间 Pod 通信的网络结构。

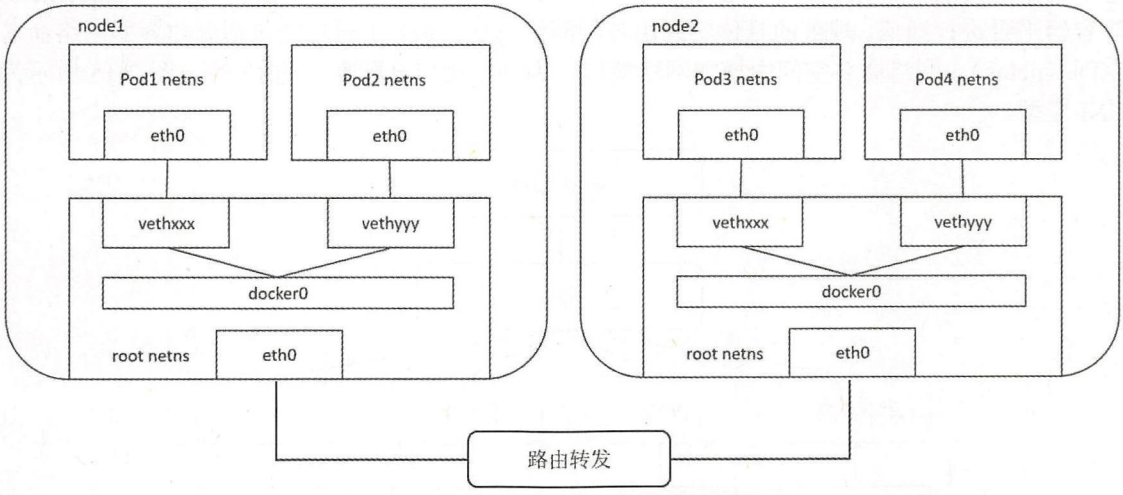


图 2-14　不同节点间 Pod 通信的网络结构

这里分析一下 Pod1 数据包如何被 Pod4 接收。

- 与上一节流程类似，Pod1 数据包传送到 docker0 网桥上。
- docker0 网桥发起 ARP 请求找到目标地址。由于 node1 上没有 Pod4 的 IP 地址，因此数据包被转发到 root netns 的 eth0。
- 外部路由通过 Pod4 的 IP 地址，将数据包转发到 node2 上的 root netns 的 eth0。
- root netns 的 eth0 再将数据转发到 docker0 网桥，然后与上一节流程类似，数据被 Pod4 接收。

现有许多开源网络方案提供了实现不同节点 Pod 之间通信的能力，帮助 Kubernetes 增加网络能力，同时也提供了部分的网络策略能力，后续内容中会针对开源网络方案进行详细讲解。

2.4.2　Kubernetes 网络模型

针对上节中 Pod 中容器之间通信和同一节点 Pod 之间通信，在 Linux 提供的 Network NameSpace 和 Docker 网络的基础上都能很好地解决。Kubernetes 网络的核心难点是解决跨节点网络不通的问题。导致跨节点网络不同的原因主要是：

- 容器 IP 地址重复。在各个节点上的容器 IP 地址是由 Docker 等工具自动分配 IP 的，可能造成重复。
- 容器地址不可达。对于不同的节点以及网络上的路由设备之间，并不知道这些 IP 容器网段的地址是如何分配的，因此数据包即使被发送到了网络中，也会找不到路由而被丢掉。

为了解决跨节点网络不通的问题，Kubernetes 网络需要满足两个基本原则：

- Kubernetes Pod IP 为 IP per Pod 模式，即每个 Pod 都具有唯一确定的 IP，避免 Pod 之间的 IP 冲突。
- Pod 之间组成的网络是一个互联的、扁平的网络拓扑。访问者不需要额外建立到 Pod 之间的连接。

目前不同的容器平台（Kubernetes、Swarm 等）都需要网络方案，为了对网络方案进行规范，Google 和 CoreOS 公司主导制订了容器网络标准 CNI（Container Network Interface）。CNI 本身并不是网络的具体实现，而是一种标准规范。CNI 的制订是在 Rkt 网络提议之上，综合考虑了网络的灵活性、扩展性、IP 分配、多网卡等因素得来的，通过 JSON 格式的文件与 Kubernetes 等容器平台进行通信。规范的具体实现由各种网络插件完成，主要功能包括创建容器网络命名空间（netns）、网络命名空间中增加网络接口、为网络接口分配唯一的 IP 等。图 2-15 所示为 CNI 模型。

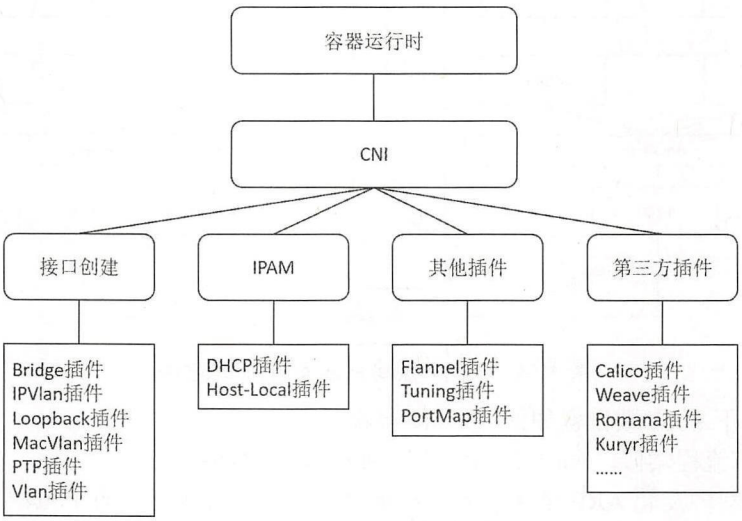


图 2-15　CNI 模型

图中接口创建、IPAM、其他插件是官方提供的网络插件方案，同时还有 Calico 等第三方网络插件方案。结合上面给出的 CNI 模型，讲述一下 Pod 网络的创建过程：首先 kubelet 在创建 pause 容器并生成网络后调用 CNI 驱动，CNI 驱动调用具体实现插件，CNI 插件为 pause 容器进行网络配置，然后 Pod 中其他容器共享 pause 的网络资源。

2.4.3　Kubernetes 开源网络方案

CNI 只是提供了一种规范，现在出现了很多开源网络方案。这些网络插件因为实现模式的不同，对应网络性能、应用场景也会有所不同。现在主流的开源网络方案主要有：Flannel、Calico、Weave、Contiv、SR-IOV、Romana、OpenContrail、Kuryr 等。本书以使用最为广泛的 Flannel 和 Calico 为例进行讲解。

1. Flannel 网络

Flannel 是 CoreOS 团队针对 Kubernetes 研发的三层网络方案，帮助快速、易用的方式配置实现 Kubernetes 网络。Flannel 的核心思想是：首先指定一个网段用于分配，每个节点会占据网段中的一个子网，然后节点上的容器 IP 就会从子网中获取；并通过底层的网络方案实现跨节点三层路由。在 Flannel 的核心思想下，能有效地满足 Kubernetes 网络的两个基本原则。在后端的网络方案实现中，Flannel 提供了基于 UDP、VxLAN 和 host-gw 的方式。在各个节点上运行 flanneld 的 agent 程序，用于维护节点上的网络分配。

图 2-16 所示为以 UDP 后端为例的 Flannel 网络模式，UDP 后端是覆盖网络（Overlay）的方式实现，即报文通过 UDP 的封装作为 payload 再进行发送，接收方接收到 UDP 包之后进行解包获取报文信息。

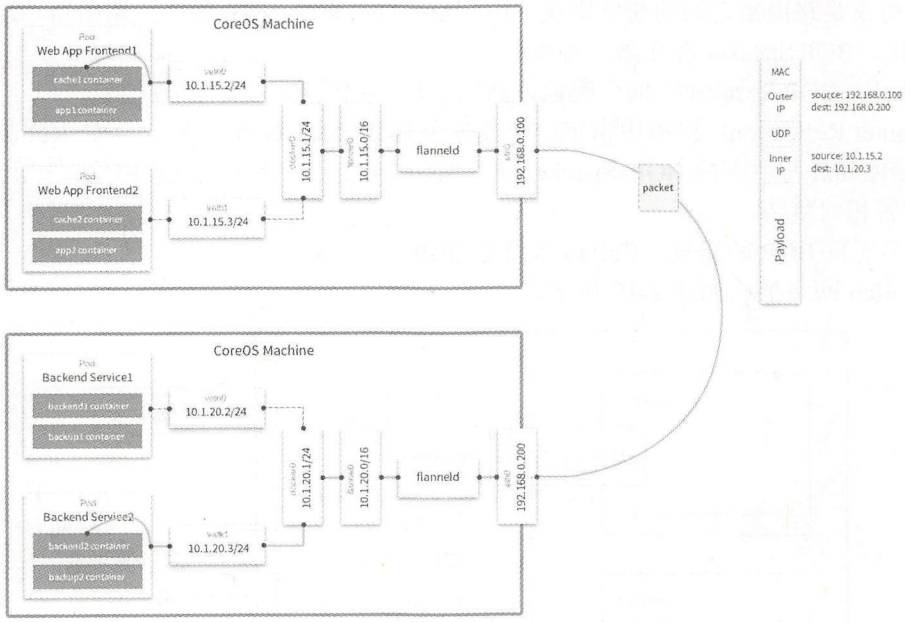


图 2-16　以 UDP 后端为例的 Flannel 网络模式

图中具体包是怎么从一个容器发到另一个容器中的呢？

Pod 发出的数据包到达 docker0 网桥。然后 docker0 网桥将数据转发到 flannel0 网卡上。flannel0 将数据发送到 flanneld 进程中。flanneld 通过 etcd 维护了各个节点之间的路由表，根据报文目的地址得到目的节点信息，然后将报文封装到 UDP 中，并发送到节点的 eth0 网卡上。目的节点收到 UDP 包之后，通过 flanneld 进行解包，然后通过 flannel0→docker0→Pod 容器接收。

Flannel 覆盖网络的推荐支持 UDP 和 VxLAN 两种方式。UDP 使用了 Flannel 自定义的一种包头协议，前面数据包在流向的过程中知道数据包会从 Linux 内核进入 flanneld 用户态进行封包和解包，因此当数据在发送和接收过程中，需要经过两次内核态到用户态再到内核态的转换，这个过程中存在着性能的损耗，并且当网络流量较大时，flanneld 会成为瓶颈。VxLAN 使用了 Linux 内核内置的 VxLAN 模块，虽然它的封包结构比 UDP 方式更复杂，但由于所有数据的封包、解包过程都在内核中完成，实际的网络性能要比 UDP 方式好很多。在 Kubernetes v1.5 版本中默认采用了 VxLAN 的方式。需要说明的是，在 Linux 较早内核版本中没有包含 VxLAN 模块，只能采用 UDP 覆盖网络的方式。

上述两种方案都是采用覆盖网络的方式，Flannel 后端还提供了 host-gw 的方式。host-gw 不采用覆盖网络，完全采用网络路由的方式，即 flanneld 会将容器网络的路由信息写到节点的路由表中，这样当需要访问某个 Pod 地址时会直接路由到 Pod 所在节点。这种方式没有引入封包、解包的额外操作，并且传输也是包数据本身。在网络规模较小的场景下，性能相对覆盖网络性能会更加优异。然而，host-gw 方式只能适用于二层可达的网络，如果网络中包含路由设备会导致数据包无法送达的情况，并且也存在网络风暴等问题。

2. Calico 网络

Flannel 网络方案中存在覆盖网络方案和路由方案 host-gw。因为 host-gw 只针对节点修改路由配置，相当于节点中存在着一个虚拟路由器，不能跨二层网络。Calico 网络主要采用的是 BGP 协议，能实现节点上的路由信息传播到节点外的设备中，从而实现了三层网络上的通信。

BGP 协议是路由器之间的通信协议，用于 AS（自治系统，拥有独立的路由器、交换机等）的相互连接。BGP Speaker 存在着两种模式。

- Mesh：BGP Speaker 之间全互联，适用于较小规模网络，扩展性较差。
- Router Reflection：网络中指定一个或多个 BGP Speaker 作为 Router Reflection，Router Reflection 与所有的 BGP Speaker 建立 BGP 连接。适用于大规模网络，扩展性较好，配置相对复杂。

对于不支持 BGP 的场景，Calico 也提供 IPIP 网络模式。

1）Calico 网络架构如图 2-17 所示。

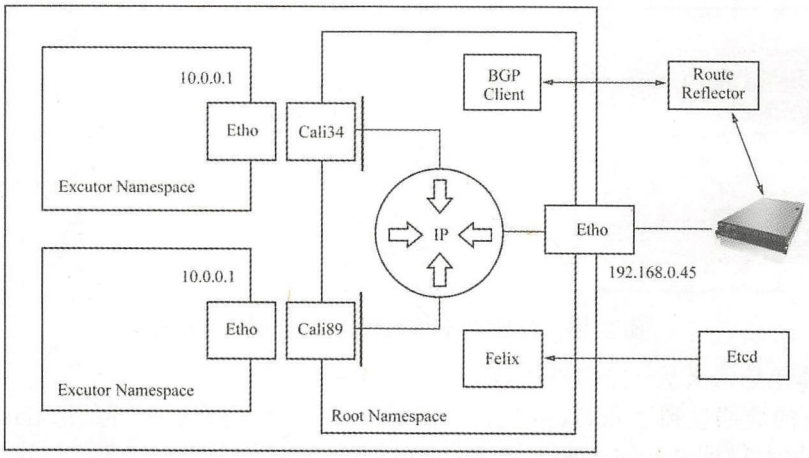


图 2-17　Calico 网络架构

Calico 核心组件包括：Felix、etcd 和 BIRD。

- Felix：在 Kubernetes 节点上运行的 Calico agent 程序，主要作用是通过 etcd 中的网络信息往节点上配置路由及 ACLs 等信息。
- etcd：负责存储网络数据，确保 Calico 网络路由等配置的准确性，通常为 Kubernetes 中 etcd 共用。
- BGP Client（BIRD）：用于将 Felix 写入 Linux 内核的路由信息传播到 Calico 网络。
- BGP Route Reflector（BIRD）：Route Reflector 与所有的 BGP Speaker 建立 BGP 连接，通过一个或者多个 BGP Route Reflector 完成集中式的路由分发。

2）Calico 网络模式。与 BGP 协议模式对应，Calico BGP 网络也存在着 Mesh 和 Router Reflection 两种模式。当大规模部署时，通过一个或者多个 BGP Route Reflector 完成集中式的

路由分发，网络成形状结构；小规模部署可以 Mesh 模式，网络成网状结构；IPIP 模式，把一个 IP 数据包又套在一个 IP 包的覆盖网络，用于公有云等不支持 BGP 的场景。

设置 Mesh 模式步骤。

```
calicoctl config set nodeTonodeMesh on #开启全互联模式
```

设置 Router Reflection 模式步骤。

```
# 关闭 Mesh 模式，再把 RR 作为 Global Peers 添加到 calico 中，calico 网络就切换到了 RR 模式
calicoctl config get asnumber #查看 AS 号
calicoctl config set asnumber 64512 #修改 AS 号
calicoctl config set nodeTonodeMesh off #关闭全互联模式

# 增加 Global Peers
$ cat << EOF | calicoctl create -f -
apiVersion: v1
kind: bgpPeer
metadata:
  peerIP: 172.16.0.10
  scope: global
spec:
  asNumber: 64567
EOF

# 查看 peer
$ calicoctl get bgpPeer --scope=global
SCOPE    PEERIP         NODE   ASN
global   172.16.0.10          64567
```

设置 IPIP 模式步骤。

```
$ calicoctl apply -f - << EOF
apiVersion: v1
kind: ipPool
metadata:
  cidr: 172.16.0.0/16
spec:
  ipip:
    enabled: true
    mode: always/cross-subnet
  nat-outgoing: true
EOF
```

3）Calico 网络优势与不足。这里讨论的 Calico 优势和不足主要针对的是 BGP 模式。

Calico 优势：

- 没有隧道封装的网络开销。
- 相比于通过 Overlay 构成的大二层层叠网络，用 iBGP 构成的扁平三层网络扩展模式更符合传统 IP 网络的分布式结构。
- 不会对物理层网络的二层参数如 MTU 引入新的要求。
- 相比 Flannel 网络，Calico 支持 Kubernetes NetworkPolicy。

Calico 不足：

- 不容易与其他基于主机路由的网络应用集成。
- Calico 的缺点是路由的数目与容器数目相同，非常容易超过路由器、三层交换，甚至节点的处理能力，从而限制了整个网络的扩张。
- Calico 的每个节点会设置大量的 iptables 规则，路由、运维、排障的难度大。
- Calico 的原理决定了它不可能支持 VPC，容器只能从 Calico 设置的网段中获取 IP。
- Calico 目前的实现没有流量控制的功能，会出现少数容器抢占节点多数带宽的情况。

- Calico 的网络规模受到 BGP 网络规模的限制。

4）Calico NetworkPolicy 支持。在 Kubernetes v1.3 中引入了 NetworkPolicy，NetworkPolicy 采用基于策略配置实现网络访问的控制。Calico 网络支持 Kubernetes 的 NetworkPolicy 功能，下面展现了 Kubernetes NetworkPolicy 的基本配置方式。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

上面过程配置的网络策略为：

- 在 default Namespace 中隔离了 label 包含 role=db 的 Pod。
- 允许 IP 地址 CIDR 为 172.17.0.0/16 且不包括 172.17.1.0/24 的所有 Pod，连接 default Namespace 中 label 包含 role=db 的 Pod 的 TCP 6379 端口。
- 允许 label 包含 role=frontend 的 Pod，连接 default Namespace 中 label 包含 role=db 的 Pod 的 TCP 6379 端口。
- 允许在 label 包含 project=myproject 的 Namespace 中的所有 Pod，连接 default Namespace 中 label 包含 role=db 的 Pod 的 TCP 6379 端口。
- 允许 default Namespace 中 label 包含 role=db 的 Pod 访问 IP 地址 CIDR 为 10.0.0.0/24 的所有 Pod 的 TCP 5978 端口。

2.5 Kubernetes 高级特性

前文介绍了 Kubernetes 中 Pod、Service 等基础资源对象以及其基本的使用特性。但在一些特定场景下，Kubernetes 的基本特性往往满足不了实际需求。例如，Kubernetes 多集群的管理、GPU 资源的支持，特定需求的应用在 Kubernetes 上分布式配置管理等。本部分讲解

Kubernetes 中的高级特性对这些需求的支持。

2.5.1 Federation

Federation（Kubernetes 联邦）是从 Kubernetes v1.3 版本引入的，目标是对多个 Kubernetes 集群进行统一调度管理。帮助企业进行快速有效的、低成本的跨区跨域、跨不同的云平台的集群管理。

Kubernetes 为什么要引入 Federation 呢？主要有以下四点。

- 避免厂商的锁定：为用户提供多云、混合云场景，在这种场景下，多个云厂商的支持也避免了单一云厂商的锁定。
- 应用/Kubernetes 集群高可用：Federation 提供的多集群的管理，提供了 Kubernetes 的高可用，当一个 Kubernetes 无法使用的时候，仍有其他 Kubernetes 集群可以使用，在这个基础上提供了应用服务层面的高可用。
- 跨集群服务发现：Federation 为多个 Kubernetes 集群提供了跨集群的服务发现。
- 跨集群资源同步或调度：Federation 提供在多个 Kubernetes 中的调度策略，多集群间的 Pod 副本平衡，以及能监控各个 Kubernetes 集群资源是否达到期望的状态。

1. Federation 架构

如图 2-18 所示，可以看出 Federation 联邦管理 IDC、阿里云和 AWS 三个 Kubernetes 集群。Federation 由 Federation Controller Manager、Federation API Server 和 etcd 三个核心组件组成，这些核心组件的作用与 Kubernetes 本身比较类似。

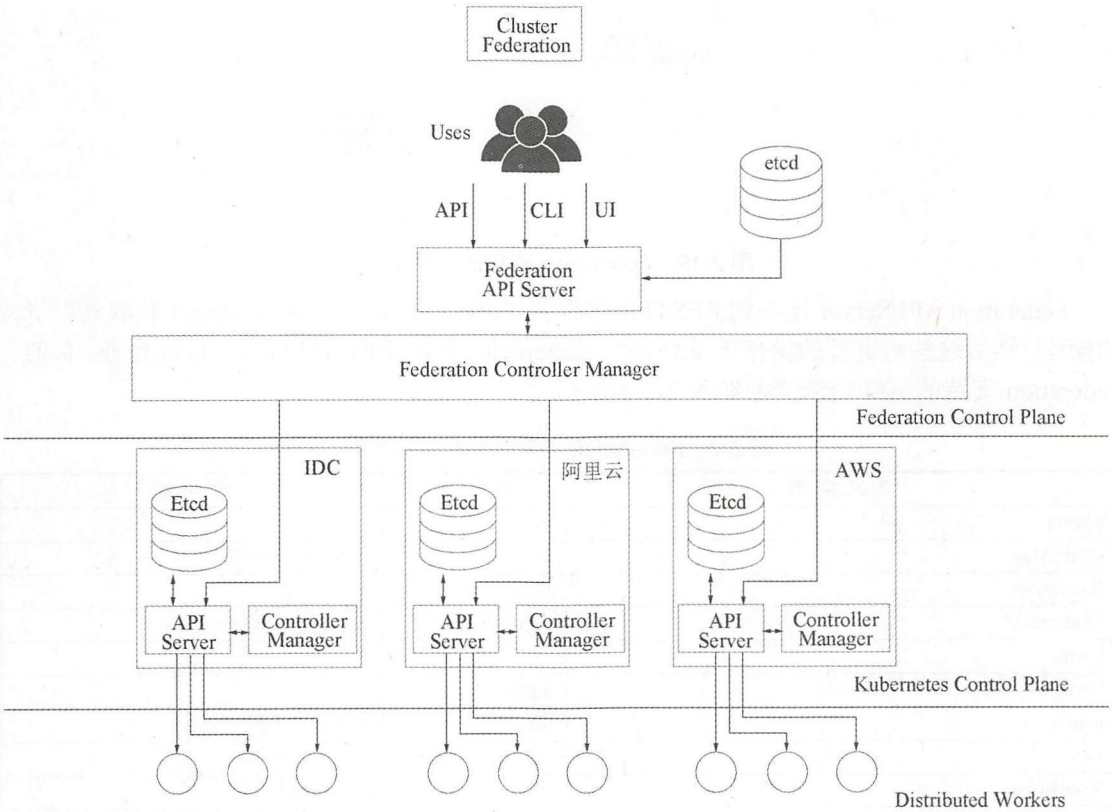


图 2-18 Federation 架构

- Federation API Server 提供各类资源对象的 RESTful 接口，负责模块间的通信。
- Federation Controller Manager 作为集群的控制模块，通过 Federation API Server 获取联邦集群的状态，然后下发各个集群的 API Server 完成集群管理。
- etcd 主要用于数据的存储。联邦控制面板将它的数据存储在其中。

除了上述几个组件，DNS 在 Kubernetes Federation 中也十分重要，Federation 提供了自动配置的 DNS 服务器，确保全局的 VIP 或者 DNS 记录可以访问部署在多个 Kubernetes 集群中的后端服务。提供了跨集群服务发现能力，也保证系统的高可用。目前，Federation 支持的 DNS Provider 有 AWS/route53、Azure/azuredns、Coredns 和 Google/clouddns。

2. Federation 资源

图 2-19 所示为 Federation 资源的创建流程。

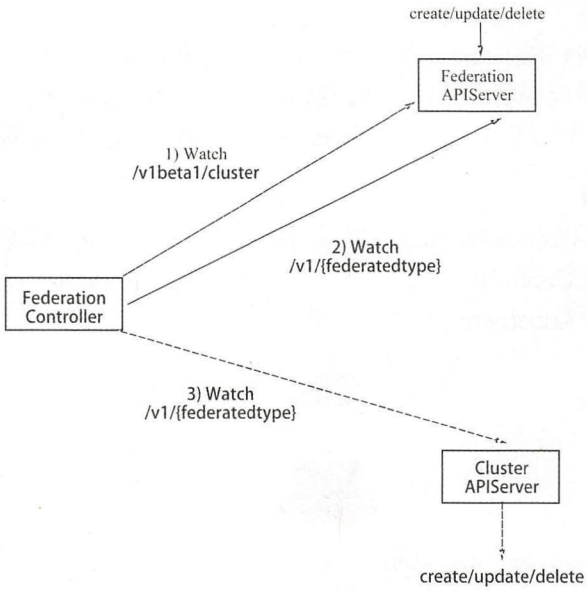


图 2-19　Federation 资源的创建流程

Federation API Server 接收到 RESTful 的请求，Federation Controller Manager 获取到对资源的操作，然后这些将资源的操作下发到各个 Kubernetes 集群中的 API Server 进行处理。目前，Federation 支持的 API 资源类型如表 2-2 所示。

表 2-2　Federation 支持的 API 资源类型

资 源 类 型	版　本	说　明
Cluster	v1	
ConfigMap	v1	不支持级联删除
DaemonSet	beta1	
Deployment	beta1	不支持 rollout 等
Events	v1	
Horizon Pod Autoscalers（HPA）	alpha	
Ingress	beta1	暂支持 Google Cloud
Jobs		不支持级联删除
Namespaces	v1	不支持级联删除
ReplicaSets	beta1	不支持级联删除
Secrets	v1	不支持级联删除

3. 部署 Federation 平面

这里有 hangzhou 和 shanghai 两个 Kubernetes 集群：

```
$ kubectl config get-contexts
CURRENT   NAME        CLUSTER                 AUTHINFO        NAMESPACE
*         hangzhou    kubernetes-hangzhou     admin-hangzhou
          shanghai    kubernetes-shanghai     admin-shanghai

# 在 hangzhou 集群的 master 节点安装 kubefed
$ curl -L http://aliacs-k8s-cn-hangzhou.oss.aliyuncs.com/bin/kubefed-1.8.4 -o kubefed
$ cp kubefed /usr/bin
$ chmod +x /usr/bin/kubefed

# 初始化 Federation 平面
$ kubefed init federation \
            --host-cluster-context=hangzhou \
            --dns-provider="alidns" \
            --dns-zone-name="opcnet.cn." \
            --dns-provider-config="alidns.yaml" \
            --image="registry.cn-hangzhou.aliyuncs.com/google-containers/hyperkube-amd64:
v1.8.4-4_99c084ce" \
            --etcd-image='registry.cn-hangzhou.aliyuncs.com/google-containers/etcd-amd64:
3.1.11' \
            --etcd-persistent-storage=false

Creating a namespace federation-system for federation system components... done
Creating federation control plane service..... done
Creating federation control plane objects (credentials, persistent volume claim)... done
Creating federation component deployments... done
Updating kubeconfig... done
Waiting for federation control plane to come up.....................................done
Federation API server is running at: 39.15.186.2

# 添加 hangzhou、shanghai 两个 Kubernetes 集群到 Federation 中
kubefed join hangzhou \ #加入联邦的集群命名名字
            --context=federation \ #联邦的 context
            --cluster-context=hangzhou \ #要添加集群的 context
            --host-cluster-context=hangzhou #主集群的 context
kubefed join shanghai \
            --context=federation \
            --cluster-context=shanghai \
            --host-cluster-context=Hangzhou

$ kubectl get cluster --context=federation
NAME       STATUS    AGE
hangzhou   Ready     2m
shanghai   Ready     52s
```

4. 集群间 Pod 副本平衡

可以通过 Federation ReplicaSet、ClusterSelector Annotation 和 Federation HPA 三种方式完成集群间 Pod 副本的平衡。

Federation ReplicaSet 采用 Federation 中的 ReplicaSet。主要用法与 Kubernetes 本身的 RS 用法相似。

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: demo
  lables:
    app: demo
  annotaions:
    federation.kubernetes.io/replica-set-preferences:
```

```
        {
          "rebalance": true,
          "clusters": {
             "test1": {
              "minReplicas": 1,
              "maxReplicas": 10,
              "weight": 1
             },
             "test2": {
              "minReplicas": 3,
              "maxReplicas": 10,
              "weight": 2
             }
          }
        }
    spec:
      replicas: 10
      template:
        metadata:
          labels:
            app: demo
        spec:
          containers:
            - name: fed-demo
          image: demo:latest
```

minReplicas、maxReplicas 指定了 test1 和 test2 集群的最小 Pod 数量和最大 Pod 数量，以及 weight 指定了两个集群 Pod 数量的权重值。

ClusterSelector Annotation 可以根据 ClusterSelector 指定 Pod 调度的集群：

```
    metadata:
      annotations:
        federation.alpha.kubernetes.io/cluster-selector: '[{"key": "pci", "operator": "In", "values":
["true"]}, {"key": "environment", "operator": "NotIn", "values": ["test"]}]'
```

Federation HPA 可以根据集群的 CPU 负载等因素水平扩展集群中的 Pod 数量。

```
    apiVersion: autoscaling/v1
    kind: HorizontalPodAutoscaler
      metadata:
        name: php-apache
    spec:
      scaleTargetRef:
        apiVersion: apps/v1beta1
        kind: Deployment
        name: php-apache
      minReplicas: 1
      maxReplicas: 10
      targetCPUUtilizationPercentage: 50
```

2.5.2 GPU 支持

GPU 资源在机器学习等应用的场景下有着非常重要的需求。在 Kubernetes v1.3 中提供了对 NVIDIA GPU 实验性的支持，在 v1.6 中全面对 NVIDIA GPU 的支持，可以识别节点中的所有 GPU 资源。

1. 部署 NVIDIA GPU 设备插件

Kubernetes 在使用 GPU 资源前需要进行下述操作启动 GPU 资源的支持。在 Kubernetes v1.8 开始采用设备插件（Device Plugins）的方式使用 GPU 资源。

下面是 Kubernetes 上部署 NVIDIA GPU 设备插件预先步骤：

1）Kubernetes 节点需要预先安装好 NVIDIA 驱动，否则 kubelet 将无法获取 GPU 信息；

如果节点的 Capacity 属性中没有 NVIDIA GPU 的数量，则可能是 NVIDIA 驱动没有安装或者安装失败。

2）Kubernetes 中 API Server 和 kubelet，feature-gates 里面 DevicePlugins 参数必须设置为 true：--feature-gates="DevicePlugins=true"，在 Kubernetes v1.10 中则不需要进行配置。

3）Kuberntes 节点需要安装 nvidia-docker 2.0。

4）nvidia-container-runtime 需要代替 runc 成为 docker 的默认运行时。

部署 NVIDIA GPU 设备插件：

```
# For Kubernetes v1.10
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.10/nvidia-device-plugin.yml
```

2. 容器使用 GPU 资源

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: cuda-vector-add
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1 # requesting 1 GPU
```

上述 Pod yaml 指定了 resource.limits.nvidia.com/gpu:1 表明 Pod 资源需要一块 GPU 资源。容器使用 GPU 资源需要注意的是：

- GPU 资源必须在 resources.limits 中请求，resources.requests 中无效。
- 容器可以请求一个或多个 GPU，不能只请求一部分。
- 多个容器之间不能共享 GPU。

对于不同节点上包含不同型号的 NVIDIA GPU。可以使用 Node Selector 将 Pod 调度到所需要型号的 NVIDIA GPU 的节点上。

3. TensorFlow On Kubernetes

TensorFlow 是 Google 公司开源的机器学习框架，手动构建大规模的 TensorFlow 集群时并不容易，TensorFlow On Kubernetes 将帮助解决这个问题。TensorFlow 进行机器学习过程中需要 GPU 资源，这里主要讲解在 Kubernetes 上部署 TensorFlow，并且 Kubernetes 为 TensorFlow 提供 GPU 资源。

TensorFlow 中 PS（参数服务器 Parameter Server）采用 Kubernetes Deployment 部署。TensorFlow 中 Worker（用于计算任务）采用 Kubernetes Job 部署，Worker 训练正常完成退出，就不会再重启容器了。因此，Job 中的 Pod 的 restartPolicy 要设置成 Never 或者 OnFailure。下面部署一个分布式 TensorFlow 核心过程：

```
# PS 以 Deployment 部署在 Kubernetes 中
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ps0
spec:
  replicas: 1
  template:
    metadata:
```

```
      labels:
        app: ps0
        role: ps
    spec:
      containers:
      - name: ps0
        image: tensorflow/tensorflow:1.0.1-gpu
        ports:
        - containerPort: 2222
resources:
          limits:
            nvidia.com/gpu: 1
        command: ["/bin/sh", "-c", "..."]
---
apiVersion: v1
kind: Service
metadata:
  labels:
    name: tf-ps0-service
  name: ps0
spec:
  selector:
    app: ps0
  ports:
  - port: 2222
    targetPort: 2222

# worker 以 Job 部署在 Kubernetes 中
apiVersion: batch/v1
kind: Job
metadata:
  name: worker0
spec:
  template:
    metadata:
      labels:
        app: worker0
        role: worker
    spec:
      containers:
      - name: worker0
        image: tensorflow/tensorflow:1.0.1-gpu
        ports:
        - containerPort: 2222
        resources:
          limits:
            nvidia.com/gpu: 1
        command: ["/bin/sh", "-c", "..."]
      restartPolicy: Never/OnFailure
---
apiVersion: v1
kind: Service
metadata:
  labels:
    name: tf-worker0-service
  name: worker0
spec:
  selector:
    app: worker0
  ports:
  - port: 2222
    targetPort: 2222
```

2.6　Kubernetes 生态

Kubernetes 在众多容器编排管理平台中脱颖而出，不仅仅是因为其本身的强大能力，Kubernetes 生态的完善也是十分重要的原因。本部分主要介绍 Kubernetes 当下较为火热的技术，也帮助读者更加全面地了解 Kubernetes。

2.6.1　Kubernetes 包管理工具 Helm

Helm 是由 Deis 发起的一个开源工具，是 Kubernetes 中的包管理器，类似 yum 包管理工具，yum 用来安装 RPM 包，而 Helm 用来安装 charts，这里的 charts 类似于 RPM 软件包，有助于简化部署和管理 Kubernetes 应用。对于 Kubernetes 上一个复杂的应用，会有 Deployment、Service、Configmap 等许多复杂的资源配置文件，这些资源的部署变更迭代等操作带来很大的挑战，Helm 的出现可以帮助解决这些问题。

1. Helm 基本概念及架构

首先给出 Helm 的基本概念。

- Chart：一个 Helm 包，其中包含了在 Kubernetes 上运行一个应用所需要的镜像、依赖和 Kubernetes 资源定义 yaml 文件等。
- Release：在 Kubernetes 集群上运行的 Chart 的一个实例。在同一个集群上，一个 Chart 可以被多次部署。每次部署都会创建一个新的 Release，对应新的 Release 名称。
- Repository：用于发布和存储 Charts 的仓库。

Helm 的架构如图 2-20 所示。

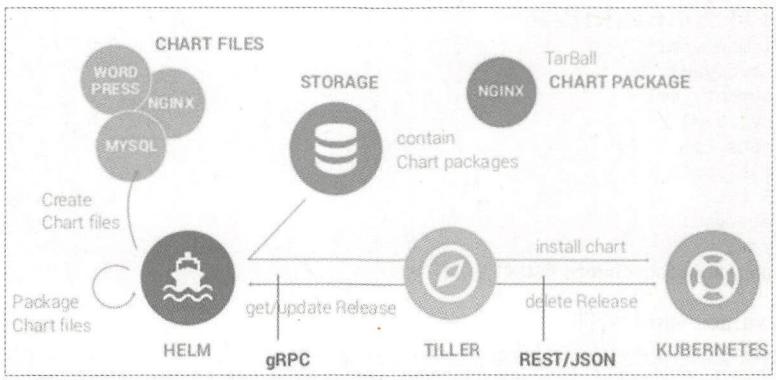


图 2-20　Helm 的架构

主要包含 Helm 两个组件 Helm Client 和 Tiller。

- Helm Client：Helm 命令行工具，使用 Golang 语言实现，通过 gRPC 协议与 Tiller 进行交互。主要负责 Charts 制作、Repository 管理、向 Tiller 发送请求等功能。
- Tiller：运行在 Kubernetes 中，用于与 Helm Client、Kubernetes API Server 进行交互。主要负责接收 Helm Client 请求、在 Kubernetes 上部署迭代 Charts。

2. 安装 Helm Client 和 Tiller

下面为安装 Helm Client 和 Tiller 的主要步骤。

```
# 安装 Helm Client
$ curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get > get_helm.sh
$ chmod 700 get_helm.sh
```

```
$ ./get_helm.sh

# 安装 Tiller
$ helm init

# 查看安装情况
$ kubectl get pod -n kube-system
NAME                            READY   STATUS    RESTARTS   AGE
tiller-deploy-5846bbf65f-mljjh   1/1     Running  0          3m
# helm version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
```

3. 使用 Helm 构建 GitLab 应用

首先在 Helm Repository 中搜索 GitLab 应用。

```
$ helm search gitlab-ce
NAME            VERSION DESCRIPTION
stable/gitlab-ce 0.2.1   GitLab Community Edition
```

可以看出在 Repository 为 stable 中含有 gitlab-ce 这个 Chart。为了更好地理解，特下载并查看 Chart 结构。

```
$ tree gitlab-ce/
mychart/
├── charts #依赖的 Chart
├── Chart.yaml #Chart 本身的版本和配置信息
├── README.md
├── requirements.lock
├── requirements.yaml #GitLab Charts 依赖的 Charts 信息
├── templates #配置 GitLab 模板目录
│   ├── configmap.yaml
│   ├── data-pvc.yaml
│   ├── deployment.yaml
│   ├── etc-pvc.yaml
│   ├── _helpers.tpl
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── secrets.yaml
│   └── svc.yaml
└── values.yaml #Gitlab Charts 默认的配置信息
```

修改 Chart values.yaml 文件。

```
# values.yaml 变更的字段
image: gitlab/gitlab-ce:9.4.1-ce.0
externalUrl: http://git.linkcloud.cn
gitlabRootPassword: "Gitlab123"
serviceType: NodePort
persistence:
  gitlabEtc:
    enabled: true
    size: 1Gi
    storageClass: linkcloud-rbd
    accessMode: ReadWriteOnce
  gitlabData:
    enabled: true
    size: 10Gi
    storageClass: linkcloud-rbd
    accessMode: ReadWriteOnce
postgresql:
  postgresUser: gitlab
  postgresPassword: gitlab
```

```
    postgresDatabase: gitlab
    persistence:
      storageClass: linkcloud-rbd
      size: 10Gi
  redis:
    redisPassword: "gitlab"
    persistence:
      storageClass: linkcloud-rbd
      size: 10Gi
```

部署 GitLab Chart。

```
# 部署
helm install -f values.yaml --name gitlab stable/gitlab-ce --namespace=linkcloud

# 查看部署情况
$ helm list
NAME  REVISION  UPDATED                  STATUS     CHART         NAMESPACE
gitlab  1  Wed May 2 14:58:15 2018    DEPLOYED   gitlab-ce-0.2.1  default
```

ingress 暴露服务。

采用 Nginx 作为 ingress-controller，首先 Kubernetes 环境中要具备 nginx-ingress-controller 以及 default-backend 服务。然后需要对 values.yaml 文件进行修改。

```
ingress:
  annotations:
  enabled: true
  tls:
  url: gitlab.linkcloud.cn
```

gitlab.linkcloud.cn 需要能解析到 nginx-ingress-controller 服务对应节点。完成 gitlab-ce 部署后，可以通过 http://gitlab.linkcloud.cn 访问 gitlab-ce 服务。

上述过程讲解了如何使用 Helm 构建 Gitlab 应用，由于 gitlab-ce Chart 已经存在于 stable 这个 Repository 中了，而实际中的应用需要自己制作 Chart 包。

```
# 生成默认的 chart 包
$ helm create mychart
```

然后修改 mychart 中的各类文件，制作成应用对应的 Chart，进而通过 Helm 进行部署到 Kubernetes 中。

2.6.2 Service Mesh

Service Mesh（也称为服务网络）是用于处理服务间通信的基础设施层，负责服务之间的网络调用、限流、熔断和监控。Service Mesh 的出现推动了传统的应用架构快速转向微服务或者云原生应用架构。区别于 Spring Cloud 等传统入侵式的微服务框架，Service Mesh 则采用了非入侵式的方式，对于应用的开发人员来说将无须关注 Service Mesh 这一层。图 2-21 所示为 Service Mesh 架构图。

目前 Service Mesh 的主流开源方案有：Buoyant 公司推出的 Linkerd，Google、IBM 和 Lyft 联手推出的 Istio 及 2017 年底 Buoyant 公司再次重磅推出的 Conduit。本节将以 Istio 为主体进行进一步讲解。

1. Istio 架构组件

Istio 是一个开放的平台，帮助微服务之间连接、管理和保障。通过在 Sidecar 代理的模式拦截微服务间的网络数据，然后由 Istio 的控制平面进行配置管理实现服务之间的负载均衡、认证、监控等功能。

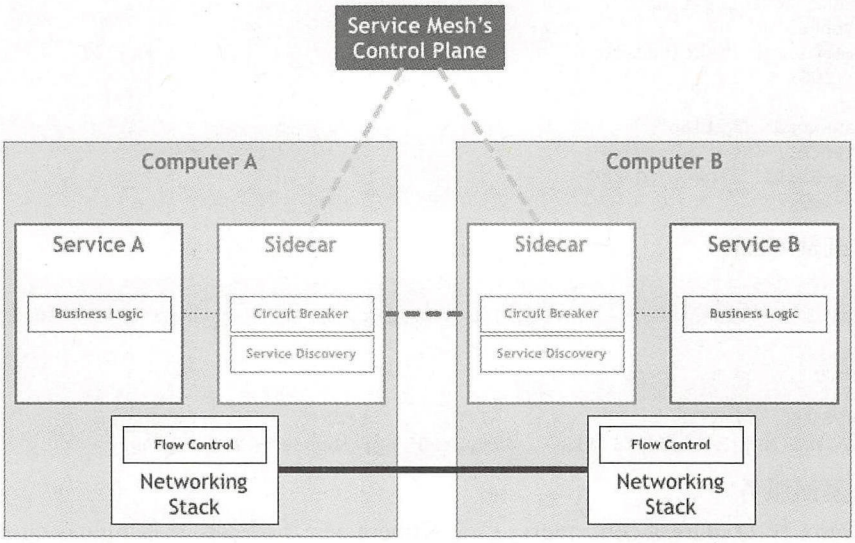


图 2-21　Service Mesh 架构图

Istio 的架构如图 2-22 所示。

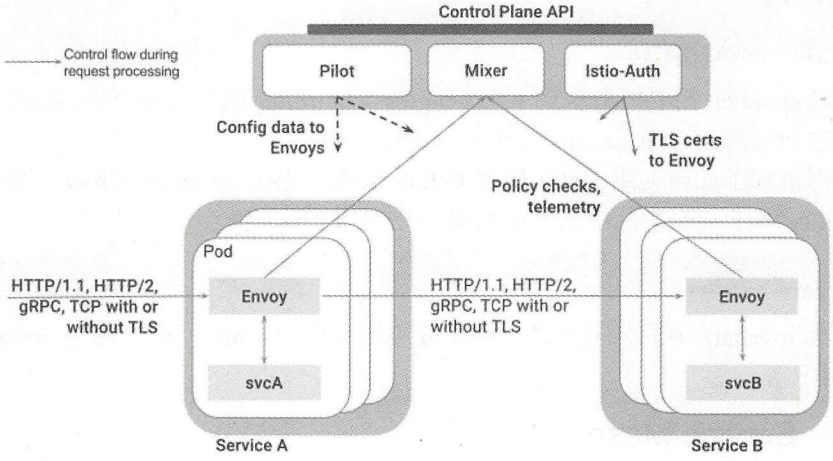


图 2-22　Istio 的架构

可以看出 Istio 架构分为控制平面和数据平面两个部分。

- 控制平面：主要负责管理和配置代理实现流量的路由转发，以及在运行时执行策略。
- 数据平面：由一组智能的代理（Envoy）部署为 Sidecar 的模式控制微服务间的网络通信。

架构中包含了 Envoy、Mixer、Pilot 和 Istio-Auth 几个核心组件。

- Envoy：Istio 使用的是 Envoy 代理的扩展版本，由 C++开发具备高性能的特性，Istio 在 Envoy 的基础上对特性进行了加强。Envoy 以 Sidecar 的方式部署在 Kubernetes Pod 中。
- Mixer：负责在服务网格上执行访问控制和使用策略，以及从 Envoy 代理和其他服务中获取遥感数据。
- Pilot：为 Envoy 提供了抽象平台的服务发现能力，以及能将控制流量行为的高级路由规则转换成特定于 Envoy 的配置。

➢ Istio-Auth：通过 TLS、内置身份和证书管理为服务之间及用户提供认证服务。

2. Istio 安装

下面介绍在 Kubernetes 上安装 Istio 的方法。

```
# 下载并解压 Istio
$ curl -L https://git.io/getLatestIstio | sh -

$ cd istio-0.7
$ export PATH=$PWD/bin:$PATH

# 安装 Istio 组件
$ kubectl apply -f install/kubernetes/istio.yaml


## 安装 webhook istio-sidecar-injector 用于自动注入 Sidecar
# 生成 kuberntes CA 签名的证书/秘钥对
$ ./install/kubernetes/webhook-create-signed-cert.sh \
    --service istio-sidecar-injector \
    --namespace istio-system \
    --secret sidecar-injector-certs

# 安装 Sidecar 注入的 configmap
$ kubectl apply -f install/kubernetes/istio-sidecar-injector-configmap-release.yaml

$ cat install/kubernetes/istio-sidecar-injector.yaml | \
    ./install/kubernetes/webhook-patch-ca-bundle.sh > \
    install/kubernetes/istio-sidecar-injector-with-ca-bundle.yaml
# 安装 stio-sidecar-injector
$ kubectl apply -f install/kubernetes/istio-sidecar-injector-with-ca-bundle.yaml


# 查看 Istio 的安装情况
$ kubectl get svc -n istio-system
NAME            CLUSTER-IP    EXTERNAL-IP       PORT(S)                AGE
istio-ingress 10.233.54.8   72.10.233.47  80:32633/TCP,443:30357/TCP  8m
istio-pilot   10.233.22.245   <none>        8080/TCP,8081/TCP           8m
istio-mixer   10.233.20.23    <none>     9091/TCP,9094/TCP,42422/TCP   8m

$ kubectl get pods -n istio-system
NAME                                  READY   STATUS    RESTARTS   AGE
istio-ca-1251496532-hp4nb             1/1     Running     0        8m
istio-ingress-581839657-gnlp6         1/1     Running     0        8m
istio-sidecar-injector-2065682486-g17hl  1/1     Running     0        5m
istio-pilot-901508202-1thwv           1/1     Running     0        8m
istio-mixer-943524069-307k3           2/2     Running     0        8m
```

3. 部署应用

这里以 Istio 自带的例子进行讲解如何使用 Istio 部署应用。

```
# 部署 sleep 应用
$ kubectl apply -f samples/sleep/sleep.yaml

$ kubectl get pod
NAME                     READY     STATUS        RESTARTS   AGE
sleep-678fb97cfd-rsgc7   1/1       Running       0          6m

# 开启 Istio sidecar 自动注入
$ kubectl label namespace default istio-injection=enabled

# kubectl delete pod sleep-678fb97cfd-rsgc7
NAME                     READY     STATUS        RESTARTS   AGE
sleep-678fb97cfd-672c5   2/2       Running       0          2m
```

```
# 关闭 Istio sidecar 自动注入
kubectl label namespace default istio-injection-

kubectl delete pod sleep-678fb97cfd-672c5
kubectl get pod
NAME                     READY     STATUS        RESTARTS   AGE
sleep-678fb97cfd-bbl72   1/1       Running       0          1m
```

上述过程可以看出 Istio Sidecar 自动注入后，Pod 中的容器会比没有开启的容器数量多一个。这个名为 istio-proxy 的容器中正运行着 Envoy。

除了未开启自动注入 Sidecar，Istio 还支持手动注入 sidecar。

```
kubectl create -f <(istioctl kube-inject -f <your-app-spec>.yaml)
```

2.6.3 Serverless

2014 年底，亚马逊 AWS 发布了函数即服务（FaaS）产品 Lambda。自此，从 FaaS 上 Severless（无服务器）架构得到了提出，并受到了越来越多的关注。IT 发展的历史从服务器到 KVM 等虚拟化到容器的演进，资源的粒度越来越小，开发人员需要关注也越偏向应用本身。Serverless 的出现也迎合着 IT 架构的发展趋势，Serverless 让开发人员只需要关注代码层面。

1）Serverless 不仅仅是 FaaS。

- FaaS 是一个计算服务，主要功能有部署、可伸缩性、执行和结算。
- Serverless 架构属于平台即服务，针对事件驱动、短暂性的工作负载。它结合了 FaaS 与其他云服务（数据库、消息队列等）构建复杂的系统，并依赖 Faas 提供的计算服务功能。

2）Serverless 具备以下特性。

- 无运维管理：部署代码无须进行预先和完成之后的操作，整个部署过程无须关注网络、监控、调度、资源等各个层面。
- 自动伸缩：无须预先写定脚本实现扩容或者缩容，平台具备根据请求负载的情况自动伸缩函数实例。
- 按需付费：传统的方式应用部署上去，无论访问量或者负载高低，都需要计费。而使用 Serverless，用户仅需要为函数真正调用到的次数和执行时间付费。
- 更高的效率：由于只关注代码本身，部署将更加快速；以函数作为实例，粒度更小，并且执行完成就被销毁，具备更短的生命周期。

3）Serverless 面临的困难。

- 无状态：函数实例执行完会被销毁，需要借助外部数据库或网络存储管理状态。
- 函数执行时间限制：例如，AWS Lambda 限制函数执行时间最长为 5 分钟。
- 启动延迟：针对应用不活跃以及对瞬时流量增长的情况延迟会更明显。
- 平台依赖：比如服务发现、监控、调试、API 网关等都依赖于 Serverless 平台提供的功能。

目前众多 Serverless 架构平台提供了构建 Serverless 架构的支持。目前公有云厂商提供的 Serverless 架构平台包括 AWS Lambda、Microsoft Azure Functions、Google Cloud Functions、IBM OpenWhisk 等；开源方案包括 Kubeless、IBM 开源的 Apache OpenWhisk、Fn、Funktion、Fission、Iron Functions 等。

下面主要讲解 Kubeless。

1. Kubeless 基础概念

Kubeless是基于Kubernetes之上的Serverless平台，依托于Kubernetes强大的功能和生态，Kubeless 在 Serverless 架构平台中备受关注，同时 Kubernetes 用户也很容易使用 Kubeless。Kubeless 采用函数实例部署、事件驱动的模式。Kubeless 使用 Kubernetes Custom Resource Definition（CRD）实现了 Kubeless 自定义的资源，Kubeless 通过运行一个控制器来管理自定义资源，用于 runtime 的启动，代码动态地注入。

Kubeless 主要的核心概念包括函数、触发器和运行时。

- 函数：函数作为 Kubeless 中独立的部署单元，区别于 Kubernetes 中微服务的部署，Kubeless 函数只是针对代码部署，在实践过程中通常将一个功能模块作为一个函数实例。函数同时包含了运行时的依赖、构建说明等信息。
- 触发器：表示作用在函数上的事件驱动源，触发器可以作用在一个函数上，也可以作用在多个函数上。当触发器被触发，Kubeless 将会调用一次相应函数。触发器具有独立于函数的生命周期。目前，Kubeless 支持的触发器包括 HTTP 触发器、Kafka 消息队列触发器、定时触发器、nats 触发器。
- 运行时：表示函数运行的代码语言类型和代码运行所需要的特定环境。目前 Kubeless 支持的语言类型包括 Python、Node.js、Ruby、PHP、Golang。

Kubernetes 上如何承载 Kubeless：

- 使用 CRD 定义函数对象。
- 每个事件驱动源封装成单独的触发器 CRD 对象。
- 独立的 CRD controller 处理 Kubeless 中 CRD 的增删改查操作。
- 使用 Deployment/Pod 运行 Kubeless 运行时。
- 使用 ConfigMap 存放函数的代码给运行时的 Pod。
- 使用 init-container 处理将函数需要的依赖。
- 使用 Service 暴露函数，使用 Ingress 将函数对外部暴露。

2. Kubeless 的安装

下面是 Kubeless 安装的过程。

```
$ export RELEASE=$(curl -s https://api.github.com/repos/kubeless/kubeless/releases/latest | grep tag_name | cut -d '"' -f 4)

$ kubectl create ns kubeless

# Kubernetes 上创建 Kubeless 资源
$ kubectl create -f https://github.com/kubeless/kubeless/releases/download/$RELEASE/kubeless-non-rbac-$RELEASE.yaml

# 查看资源创建情况
$ kubectl get customresourcedefinition
NAME                          AGE
cronjobtriggers.kubeless.io   1m
functions.kubeless.io         1m
httptriggers.kubeless.io      1m

$ kubectl get deployment -n kubeless
NAME                          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
kubeless-controller-manager   1        1        1           0          1m

# configmap 存储着 Kubeless 运行时的信息
$ kubectl get configmap -n kubeless
NAME            DATA      AGE
```

```
kubeless-config   10        2m

# 下载 Kubeless CLI
export OS=$(uname -s| tr '[:upper:]' '[:lower:]')
curl -OL https://github.com/kubeless/kubeless/releases/download/$RELEASE/kubeless_$OS-amd64.
zip && \
  unzip kubeless_$OS-amd64.zip && \
  sudo mv bundles/kubeless_$OS-amd64/kubeless /usr/local/bin/
```

3. Kubeless 实践案例

下面实现一个简单的例子。

```
# hellowithdata.py 是一个 Python 函数
vi hellowithdata.py
def handler(event, context):
    print(event)
    return event['data']

# Kubeless 部署函数
$ kubeless function deploy hello --from-file hellowithdata.py --handler hellowithdata.handler
--runtime python2.7

# 查看函数部署情况
$ kubeless function ls
NAME    NAMESPACE   HANDLER                 RUNTIME         DEPENDENCIES        STATUS
hello   default     hellowithdata.handler   python2.7       1/1                 READY

# 调用函数
$ kubeless function call hello --data 'Hello world!'
Hello world!
```

本章作者：陈京来。

CHAPTER

3

第3章 美丽联合容器云实践

本章首先介绍美丽联合集团基于 Kubernetes 和 Docker 容器云平台的技术方案、架构演进的三个阶段，以及在稳定性、效率和成本三方面所做的工作；然后介绍关键技术方案及创新点；最后谈一下个人三年多来的体会、思考和解决过的问题，以及分享一些优秀开源工具和项目。

3.1 “从零到一”：容器云平台的技术演进

美丽联合集团（以下简称“美联”）有蘑菇街和美丽说两个子品牌，另外和京东成立了一家合资子公司微选。美联的主要业务是在线女性社交电商平台，集团的虚拟化团队成立于2014年年底。美联完全从零开始建设起了集团的IaaS和PaaS平台，见证了美联从物理机、虚拟机到容器技术演进的全过程。目前，全站超过90%的业务都运行在虚拟化平台上，正朝着全容器化的目标迈进。虚拟化团队也是美联全站业务最底层的基础平台团队之一。

3.1.1 为什么要建设容器云平台

美联的前身蘑菇街成立于2010年，早期的蘑菇街是典型的LAMP架构，线上大部分用的是物理机，当时的机器是分散人工管理的，运维和监控不太完善，比如没有统一的CMDB，尤其是发布和扩容需要靠人工处理。美联从2014年年底开始建设私有云平台，希望能够实现多机房、多集群的统一资源管理，并采用虚拟化的技术提高稳定性、提升资源利用率和资源交付的效率。表3-1是2014年和2017年架构对比

表3-1 2014年和2017年架构对比

时间类别	2014年底	2017年底
资源类型	基本使用物理机，少量零散的虚拟机	KVM虚拟机和Docker容器共存，资源粒度更细，更小的资源损耗，成本更低
集群管理	分散管理，人工操作为主	多机房，多集群的统一管理，覆盖全站90%的业务，容器级业务混部
CMDB	自研了CMDB的雏形，但功能偏弱	基于CMDB的机器资源和配置管理，运维体系化平台化建设
监控告警	粒度较粗，告警不完善	完善的监控和预警，通过预警提前发现问题，稳定性大幅提升
发布	脚本完成，部署和扩容容易出错	高效、标准化的发布系统，业务镜像化

3.1.2 如何建设容器云平台

从技术的角度看，Docker容器的确有很多优势，比如它具有比KVM / Xen虚拟机更轻量化、秒级启动、简单易用、具备镜像分层机制等特点，但仅仅有容器是不够的。

建设一套完整的容器云平台，会涉及与公司的运维体系、监控系统、日志系统、基础网络环境、中间件、持久化存储等方方面面的整合，必须要从更高的维度，结合公司的发展现状、周边配套，甚至组织架构来整体设计、规划和实施，如图3-1所示。

图3-1 容器云平台涉及的架构体系

1. 技术选型背后的思考

1）投入产出比。对于创业型公司来说，建设一套完整的 PaaS 平台成本很高。起初，我们团队只有两个人，自研是不太现实的，所以主要考虑的是在开源项目的基础上进行二次开发，“站在巨人的肩膀上”，尽量选择业界成熟且可维护、可二次开发的开源项目，是更加符合实际的选择。

在较短时间内交付基本可用的产品，供小范围的业务试用，不断积累经验，提升稳定性和用户体验，这是开发 PaaS 平台的主要思路。很重要的一点，就是产品的稳定程度和业务推广的节奏是矛盾的，需要在研发上线的过程中重点关注。如果产品出现了稳定性问题，业务推广的节奏就应该放缓，甚至可以等问题解决后再推进。在这方面，我们也有过比较深刻的教训。

2）技术选型

- 容器。Docker 的易用性和学习门槛很低，社区也一直很活跃，这是选择 Docker 的原因。

如图 3-2 所示，Docker 底层的 Containerd 基于内核提供的 Cgroup 和 Namespace 实现隔离性，也就是 RunC。但随着业务的不断接入，发现 RunC 的隔离性无法满足更多业务对更强隔离性的需求。RunV，也就是类似 KataContainers 的开源项目进入了我们的视野。2018 年 5 月，Google 公司开源了基于沙盒的 gVisor，又在云计算技术圈掀起了一次技术革命浪潮。第三种是满足机器学习平台对 GPU 的容器化需求，目前 Nvidia-Container-Runtime 作为 Docker 的另一种后端，已经在为机器学习平台提供服务了。

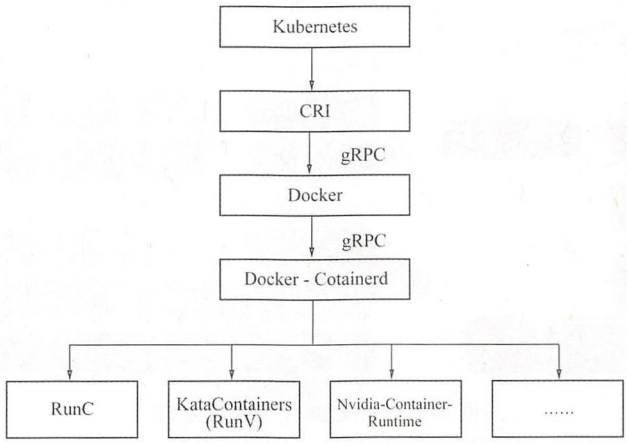


图 3-2 Docker 支持的多种后端实现

- 集群管理。Kubernetes 和 Mesos 在 2014 年年底时还不够成熟，当时 OpenStack 作为云计算领域最著名的开源项目，已经有了超过 60%的市场占有率。它的功能完备，设计理念先进，具有很好的可扩展性，方便二次开发。因此，我们选择了 OpenStack+Docker 的方案实现第一阶段的容器化。

2015 年，Google 公司发表的 Borg 论文引起了我们极大的关注，随后，业界的容器集群管理三驾马车 Kubernetes、Swarm 和 Mesos 进入了我们的视野，三者的对比如表 3-2 所示。

表 3-2 Kubernetes、Swarm 和 Mesos 的对比

对 比 项 目	Kubernetes	Swarm	Mesos
优点	Google 公司十多年运维经验，良好的设计和抽象	原生支持，简单易用	成熟，国外大公司用得较多

续表

对 比 项 目	Kubernetes	Swarm	Mesos
缺点	网络等组件需要第三方提供	不成熟，生产环境案例很少	适用场景不一样，更适合短时任务和调度
功能（集群管理、调度、网络、存储、健康检查、短任务支持、服务发现、负载均衡、CLI）	完善	仅关注集群管理，不支持短任务，健康检查等	成熟稳定，二层调度，能适配多种调度框架
学习成本	中	低	中
二次开发	容易，插件化	容易，但原始功能不够完善	复杂，Mesos 以 C++为主，Marathon 为 Scala
社区活跃度	高 腾讯，京东，乐视，滴滴，搜狐，网易，华为等	中 阿里云，新浪微博	中 爱奇艺，携程

我们从多个维度进行了深入分析和对比，最后选择了 Kubernetes 作为容器的集群管理和调度平台，并且在 2016 年完成了从 OpenStack 到 Kubernetes 的演进。

2. 整体架构

DevOps 平台整体架构如图 3-3 所示。

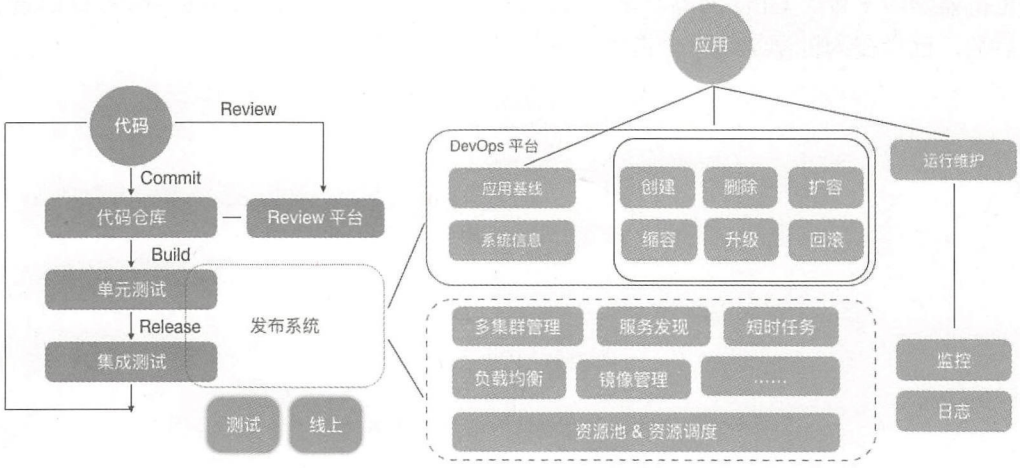


图 3-3　DevOps 平台整体架构

美联建设基于 Kubernetes 的容器云平台，目标就是希望能够提高研发人员从代码构建到业务上线的整体效率。这个过程是一个完整的 CI / CD 流程，包括从开发人员提交代码，进行代码 Review，到代码编译、单元测试、集成测试，最后构建出业务镜像，在不同的环境中部署业务，上下线服务等各个环节，形成统一的应用生命周期管理。

总体视图如图 3-4 所示。

左侧是持续集成框架，其中由应用基线管理、代码编译、单元测试、发布系统等模块组成。

中间是多集群统一管理平台。业务会分别在开发、预发和生成三个不同的环境中部署。业务的最上层是由 SLB 软负载统一管理 DNS、Nginx、LVS 等组件，将业务流量转发给不同 Kubernetes 集群中的业务容器。基于 Kubernetes 的 CaaS 由多集群管理、镜像管理、权限管理、服务发现等子模块组成。

右边是自研的监控告警系统和运维对接平台，包括监控平台、日志平台和运维 CMDB 等系统。

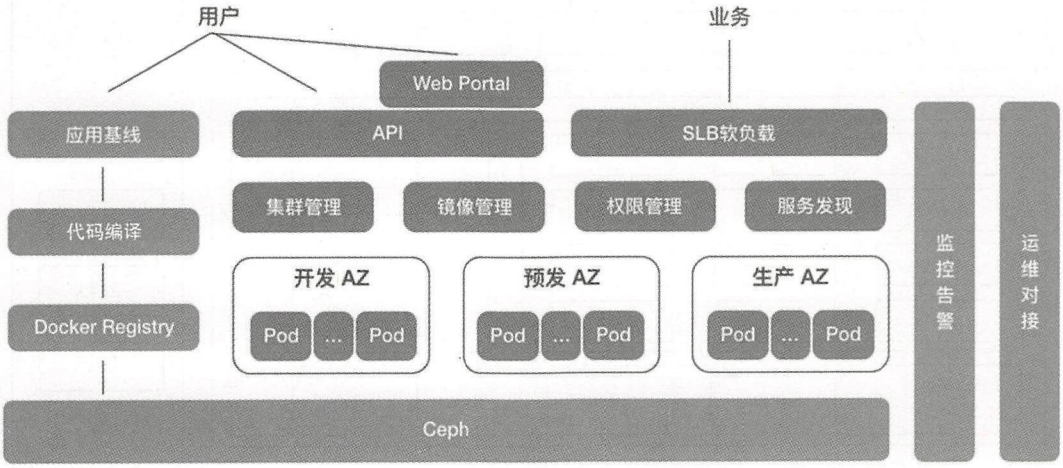


图 3-4　总体视图

3. 容器云服务

基于 Kubernetes 的容器云服务分成了四个组成部分，下面介绍它们的功能。

1）集群管理（Kube-orchestrator）：支持多机房多集群的统一管理。集群在逻辑上分成开发、预发和生产三个环境，在每个环境上通过 label 切分成不同的可用区，如图 3-5 所示。

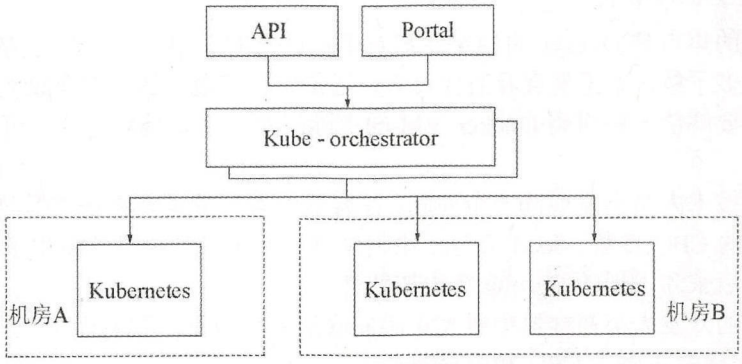


图 3-5　集群管理

2）镜像管理（Lens）：镜像管理负责镜像的访问权限控制和多机房间的镜像复制与同步。除了基本的镜像管理，还和应用配置打通。当应用配置变更后，会自动生成 Dockerfile，并触发自动生成 Docker 镜像。

3）权限管理（Privilege）：用户的容器集群由权限模块管理，从而实现多租户的隔离。权限管理和公司的 LDAP 对接，除支持认证用户的统一登录外，也支持组内一些虚拟账号的认证和操作。

4）服务发现（Radar）：服务发现通过 Headless Service 机制与 SLB 软负载中心对接，从而实现自动扩容时的实例注册和上线动作。

总的来说，容器服务包括图 3-6 所示的功能点。

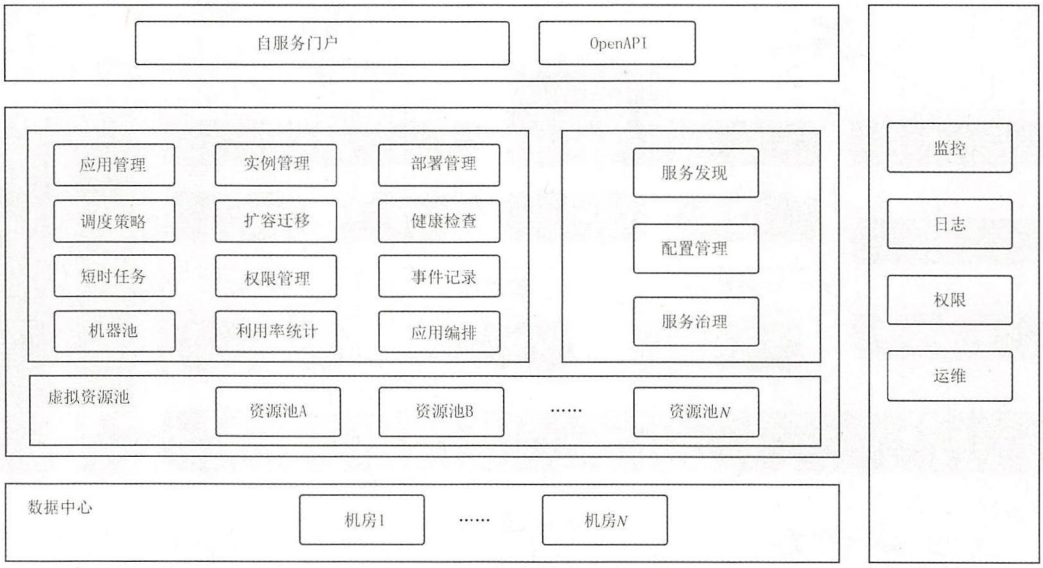


图 3-6　容器服务包括的功能点

3.1.3　架构演进

总体上说，容器云平台的演进可以分成三大阶段：容器VM化、容器镜像化和应用的标准化。

1. 阶段一：容器 VM 化

阶段一关注的焦点是 Docker 的稳定性和对用户体验的无感知。在此之前，大部分业务方对容器并没有太多了解，无论是查看监控日志，还是定位问题，都习惯登录到容器中进行。因此，这一阶段需要解决一些阻碍 Docker VM 的实际问题，主要是稳定性、可操作性和数据可视化相关的问题。

用过容器的技术人员会发现同一个问题：在容器中看到的很多系统值是物理机的数据，比如通过 top 看到的 CPU 核数、load 值等。原因是这些系统工具很早就诞生了，而它们的系统调用无法考虑到后来才出现的 Cgroup 等内核特性。

为了让应用的开发人员在容器中看到正确的底层系统数据，我们做了几方面的努力：

1）容器应用获取正确的 CPU 和内存信息

不少应用（如 JVM、Nginx）会在容器中获取 CPU 核数或内存等硬件配置信息，如果不返回容器 Cgroup 限制的正确数据，应用会在容器中收到物理机的 CPU 核数。如果应用根据物理机的 CPU 核数设置线程数量，或者进行限速等，会造成潜在的稳定性问题。

为了解决这一问题，我们实现了一个动态链接库，用 LD_PRELOAD 的方式在容器启动时加载，它的工作原理是截获用户态的 sysconf()系统调用，通过读取/proc/self/mountinfo 获取 Cgroup 路径，然后通过读取 Cgroup 数据解析容器的 CPU 和内存信息，将 CPU share 或 CPU period/quota 等值转换成 CPU 的核数。最大可用内存取自 Cgroup 中的 memory.limit_in_bytes，并转换为以页为单位的数值。glibc 中定义的 _SC_NPROCESSORS_CONF、_SC_NPROCESSORS_ONLN 两个 flag 对应 CPU 核数，_SC_PHYS_PAGES 对应页数，即内存大小。以下为 mountinfo 中一个 docker memcg 的路径信息。

```
219 214 0:29 /docker/7208cebd00fa5f2e342b1094f7bed87fa25661471a4637118e65f1c995be8a34
/sys/fs/cgroup/memory ro,nosuid,nodev,noexec,relatime - cgroup cgroup rw,memory
```

这一实现思路是从 JDK 10 中的实现借鉴而来的，可以参考博客：https://mjg123.github.io/

2018/01/10/Java-in-containers-jdk10.html。

读取 Cgroup 信息的源码：https://bugs.openjdk.java.net/secure/attachment/70120/cgstats.c。

2）容器 load 值计算。CentOS 6 和 7 自带的 Linux 内核不会计算每个容器的 load 值，而 load 值往往会作为监控告警或限流降级等的重要指标。因此在内核中实现了容器级别的 load 值计算。

已开源的阿里 v4.9 内核（https://github.com/alibaba/alikernel.git）也实现了类似的机制，git commit 是 dde3c3009e5b29b，如图 3-7 所示。

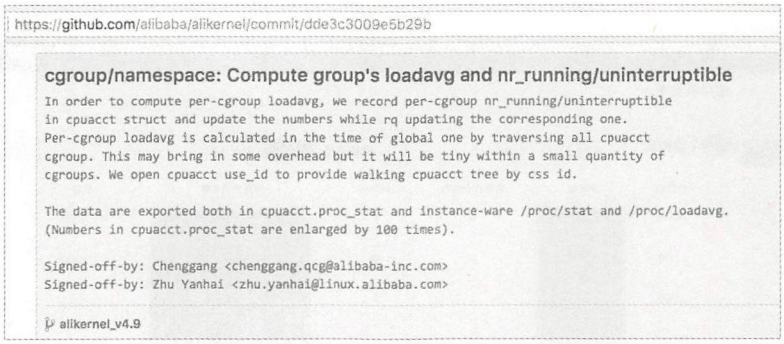


图 3-7 阿里内核中的容器 load 值计算

3）cAdvisor 接入监控系统

Kubernetes 自带的 cAdvisor 会默认从宿主机上采集大量的系统数据，这些都是准确的。把采集到的指标接入内部的监控系统 sentry，实现了容器级别的监控和报警。但是，cAdvisor 自身的 Bug 比较多，需要升级到较高的版本才比较稳定。

4）常用系统工具的替换

为了让开发人员和运维人员在容器内使用常用命令时看到的是容器的值，而不是整个物理机的指标。我们开发了一个叫 container-tools 的工具集，思路是将 top/uptime/free/iostat/df/tsar 等源码进行修改，判断如果在容器内执行，则读取/sys/fs/cgroup 中对应目录中的正确数据，如图 3-8～图 3-10 所示。

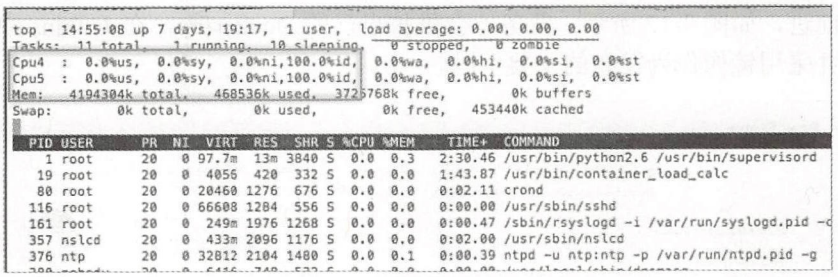


图 3-8 容器内 top 截图

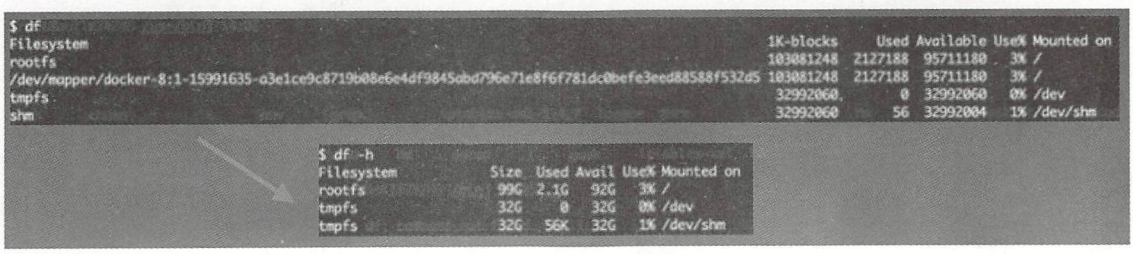


图 3-9 容器内 df 截图

```
$ tsar --cpu --mem -n1 -l
Time                ---------------------cpu---------------------  ---------------------mem---------------------
Time                user     sys    wait    hirq    sirq    util    free    used    cach   total    util
19/03/16-07:31:34   0.37     0.50   0.00    0.00    0.00    0.87    16.1G   1.6G    2.3G   20.0G    8.07
19/03/16-07:31:39   0.60     0.77   0.00    0.00    0.00    1.37    16.1G   1.6G    2.3G   20.0G    8.07
```

图 3-10　容器内 tsar 截图

5）一站式统一管理

容器 VM 化以后，可以在一个平台上同时管理 KVM 和伪装成‘VM’的 Docker。运维人员和开发人员可以一站式操作多个机房中的多个容器集群，极大地提高了生产效率，如图 3-11 所示。

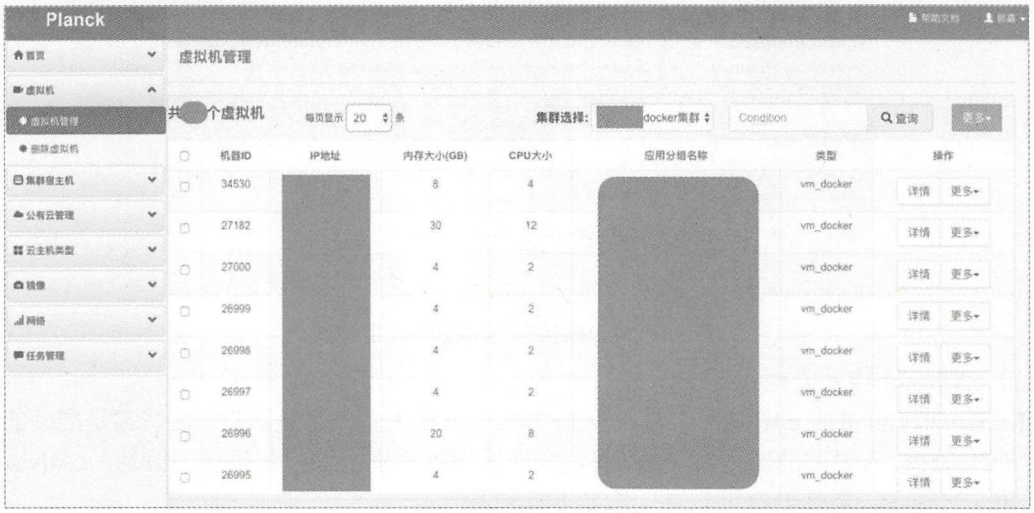


图 3-11　像用虚拟机一样管理容器

2. 阶段二：容器镜像化

第一阶段只是让业务方开始体会到了容器秒级启动、集群化高效管理的好处，但要想让业务方提高部署和发布的稳定性和效率，则离不开容器的镜像化。

这时候我们做的是向核心业务方介绍 Docker 的镜像机制，推动业务方朝着应用“微服务化”的方向前进，如图 3-12 所示。业务方会维护自己的 Dockerfile，在 Jenkins 中创建 Docker 打包任务，并采用镜像作为发布的主要手段。

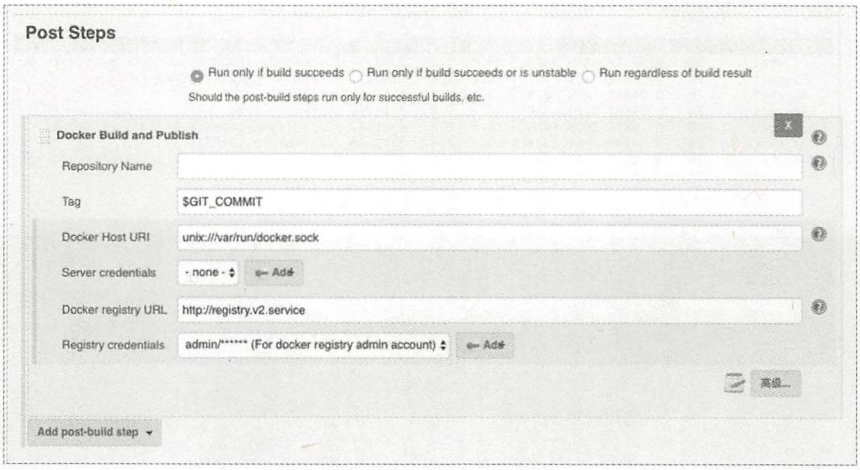


图 3-12　Docker 镜像

在这一阶段，我们提供的 UI 可以让较小的业务方自助地在界面上管理自己的集群。对于另外一部分高级用户，我们提供 RESTful API 接口，满足高级特性的需求。

这一阶段的一个关键点是要尽可能地保持上层 API 的后向兼容性，而底层设施逐步从 OpenStack+Docker 迁移到 Kubernetes+Docker 上来。经过近一年的努力，这套平台已经比较完善了，2017 年接入了包括主站的搜索、中间件、电商基础等部分有状态的业务。

图 3-13　基于 Kubernetes 的容器化集群管理

3. 阶段三：应用的标准化

经过上一阶段的建设，有部分非标准化的业务已经实现了全容器化，但这些业务尚未实现应用的标准化，需要为标准化的应用提供统一的 DevOps 化容器服务。

应用标准化的目标，是让开发人员能更容易地实现自助式的开发、部署、上线工作，业务方越来越不需要关注应用是如何部署、服务是如何被发现和接入等细节。整个环节都是自动化的，运维人员尽量少介入，甚至不介入。业务方只需要输入业务相关的源码 Git 地址、启停方式、健康检查 URL 等配置信息即可。

图 3-14 所示为基于容器的 DevOps 平台首页，罗列了该业务方所能管理的应用列表。

图 3-14　DevOps 平台首页

单击“应用详情”进入应用的详情页，这里展示了应用的具体信息，如图 3-15 所示。

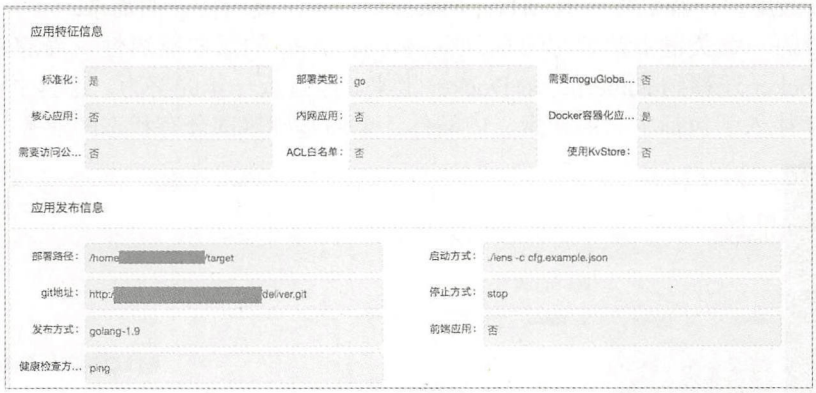


图 3-15　应用基础信息

应用基线展示了应用在开发、预发、线上等环境中的配置信息，如图 3-16 所示。

图 3-16　应用基线

应用的发布支持代码合并、灰度发布、回滚等功能，如图 3-17 所示。

图 3-17　应用灰度发布

随着 DevOps 平台稳定性和效率的不断提升，最终会实现全站的业务从虚拟机迁移到容器化平台上来，从而实现全容器化的目标。

3.1.4　稳定性、效率和成本

1. 稳定性

稳定性对于容器云这样的基础平台而言是最重要的。面对各种来自硬件故障、软件问题、安全漏洞、人为事故等不同方面的挑战，如图 3-18 所示。通过努力，我们的私有云平台经历了十几次大促活动，包括每年的“双 11”。大促期间从来没有出过任何大的事故，日常的稳定性也一直能保持在 99.95%以上。

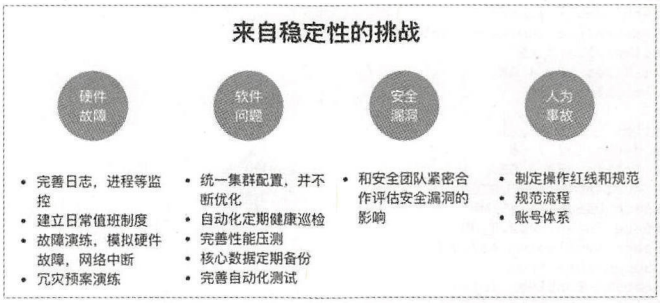


图 3-18　容器云平台面临的稳定性挑战

1）硬件故障。因为我们的容器是直接运行在物理机上的，硬件的故障会直接影响业务的可用性。而硬件的故障是没有办法避免的，因此需要通过各种技术手段或非技术手段保障业务是可用的。

针对硬件最有效的监控方式是 syslog 日志监控。比较常用的硬盘坏道、内存故障，都可以通过监控/var/log/messages 中的内核关键字提前发现问题。

我们在日志监控中增加了很多 syslog 内核关键字的监控，比如 hung_task、out of memory、segfault、Machine Check Exception。后续更好的做法是设置/etc/rsyslog.conf，把内核日志写入专门的文件中，比如/var/log/kernel.log 中，监控该文件即可，如图 3-19 所示。

图 3-19　日志监控设置

2）软件问题。除了硬件故障，更多遇到的是软件问题。解决软件问题的一个重要手段是不断完善基础软硬件的监控和针对容器的监控。

基础监控增强。Docker v1.13 以前在 CentOS6 和 CentOS7 上默认的存储后端是 Device mapper。Device mapper 实际是一个虚拟的存储设备抽象层，实际的物理存储空间可能会小于虚拟存储设备的空间，也就是存在“超配”的可能性。实时监控存储空间的使用情况就显得很有必要，如图 3-20 所示。

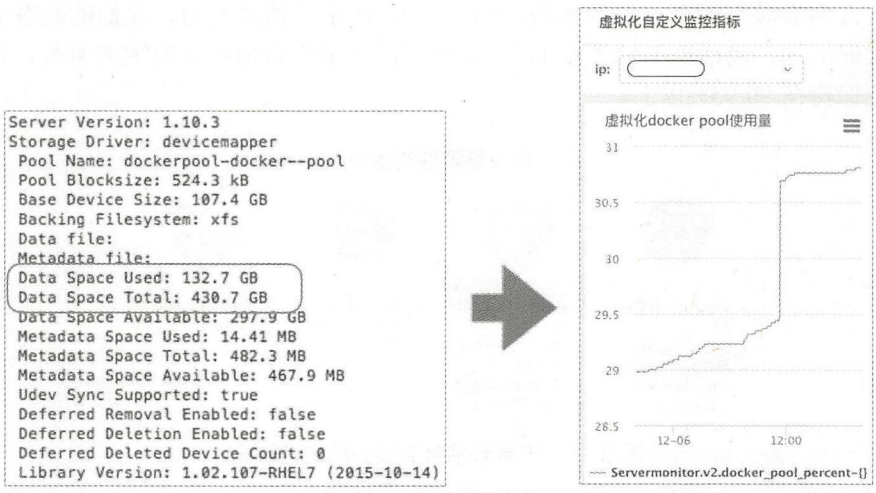


图 3-20　实时监控存储空间

➢ TCP 重传率监控。

TCP 重传率反映了一台主机的网络处理繁忙情况，通过该指标可以快速地定位很多与网络有关的疑难问题，比如网络大面积抖动、应用无故 RT 升高等。

假设每隔 1 分钟采样一次重传率，TCP 重传率的计算公式，可以写成：

$$\text{TCP 重传率} = \frac{\text{T2 重传包数-T1 重传包数}}{\text{T2 总发包数-T1 总发包数}}$$

如图 3-21 所示为 TCP 重传率。重传率是在指定时间段内，发生重传的 TCP 包占总发送包的百分比。

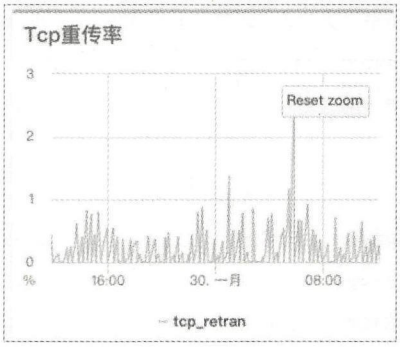


图 3-21　TCP 重传率

如果 TCP 重传率大于 1%，一般意味着网络上已经出现了网络抖动，需要进一步排查分析。

图 3-22 所示为通过 tcpretrans 工具抓取到的问题现场数据，可以直接推断出由于服务器端 10.50.185.79 上 6001 端口的应用处理能力不足，客户端 10.50.152.242 一直在尝试给服务器端

重发数据，而本质问题往往是出现在与这台客户端连接的对端服务器上。

```
10:53:18 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:19 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:20 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:21 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:22 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:23 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:24 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:25 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:26 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:27 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:28 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:29 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:30 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:31 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
10:53:32 0      10.50.152.242:34686    R> 10.50.185.79:6001    ESTABLISHED
```

图 3-22　通过 tcpretrans 工具抓取到的问题现场数据

➢ 主动事件通知

应用 oom 事件告警。我们会在计算节点部署监控的 agent，通过捕获 Docker 的事件监听 oom 的发生，然后给相应应用负责人发送告警。Docker 底层会定期监控/sys/fs/cgroup/memory/docker/docker pid/memory.oom_control 中的 under_oom 字段，发生 oom 时，内核会自动将 under_oom 标志从 0 设成 1。

```
$ cat memory.oom_control
oom_kill_disable 0
under_oom 0
```

通过命令行 docker events -f event=oom，也能观察到 oom 事件的发生。

Pod 迁移（eviction）通知。在默认情况下，Kubernetes 发生 Pod 迁移时并不会主动发出通知。我们定制了 Kubernetes 源码，对接内部 IM 工具，实现 Pod 迁移时主动通知，如图 3-23 所示。

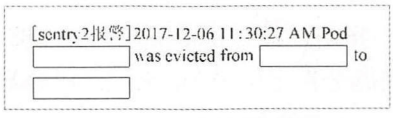


图 3-23　Pod 迁移通知

健康巡检。线上集群的配置必须统一，但由于各种升级变更、程序漏洞，甚至故障恢复，会导致各种脏数据残留在 Kubernetes 和 CMDB 里，定期的健康巡检就显得尤为必要了。

健康巡检脚本每天会自动核对 Kubernetes 和 CMDB 中的容器信息，统计搜集剩余可用计算、网络、存储资源、节点上不正常容器等信息，这样日常值班的同事可以第一时间处理，避免潜在问题的发生。

3）安全漏洞。安全漏洞是我们一直比较重视的方面，通过主动和集团安全团队合作，定期评估业界的 CVE 漏洞报告，及时修复开源组件如 openssl、dhclient 等潜在漏洞。

4）人为事故

由于经常需要进行容器云平台的发布，协助业务方进行问题定位，线上的操作难以避免，在发展的早期，也发生过一些人为的误操作。要避免这些线上事故的发生，必须要约束和加强操作流程的规范化和体系化，为此制订了专门的线上操作规范，明确了操作的红线。尤其重要的是自身发布的标准化，既能提高生产效率，又能降低线上变更导致的事故发生概率。

2. 效率

除了稳定性，效率的提升也是很重要的一方面。效率可以包含机器使用的资源效率，也就是成本，也可以包含人员投入的效率，这里主要的投入是指定位和解决线上疑难问题的工作。

除不断地增强基础监控的能力，完善主动通知能力以外，我们还做了以下事情：

1）Chatops 提升定位效率。在日常排查问题的过程中，往往需要登录 Docker 宿主机进行操作。我们在内部的 IM 工具上集成了一个小 T 自动应答机器人，背后通过 CMDB 根据 Docker IP 查询到宿主机的 IP 地址，极大地提升了排查问题的效率，如图 3-24 所示。

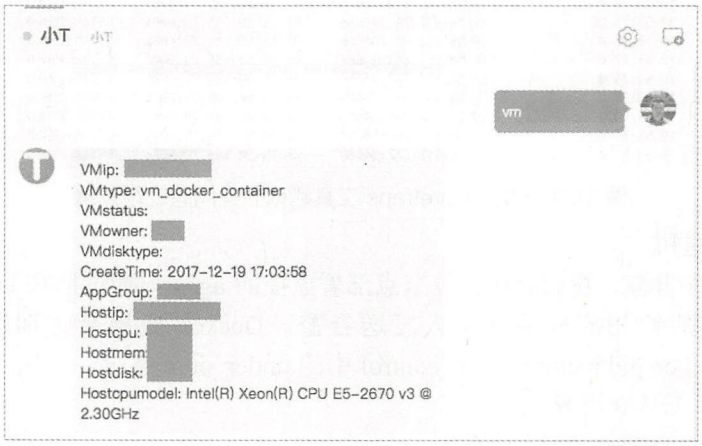


图 3-24　小 T 自动应答机器人

2）建立业务答疑群。充分利用 IM 工具的能力，通过创建答疑账号和答疑群，可以更高效地与业务方沟通，快速解决问题。

3）形成问题排查指南。在帮助业务方从系统层面定位各种业务问题的过程中，我们总结了不少经验和方法，把这些思路分门别类地整理汇总，最终形成了一个《系统问题定位排查指南》。

我们面向全公司发布了这一指南，最终是希望业务方能够自助式地快速定位系统问题。

同时，通过建设一套系统问题分析定位平台，来沉淀问题定位排查工具，如图 3-25 所示。

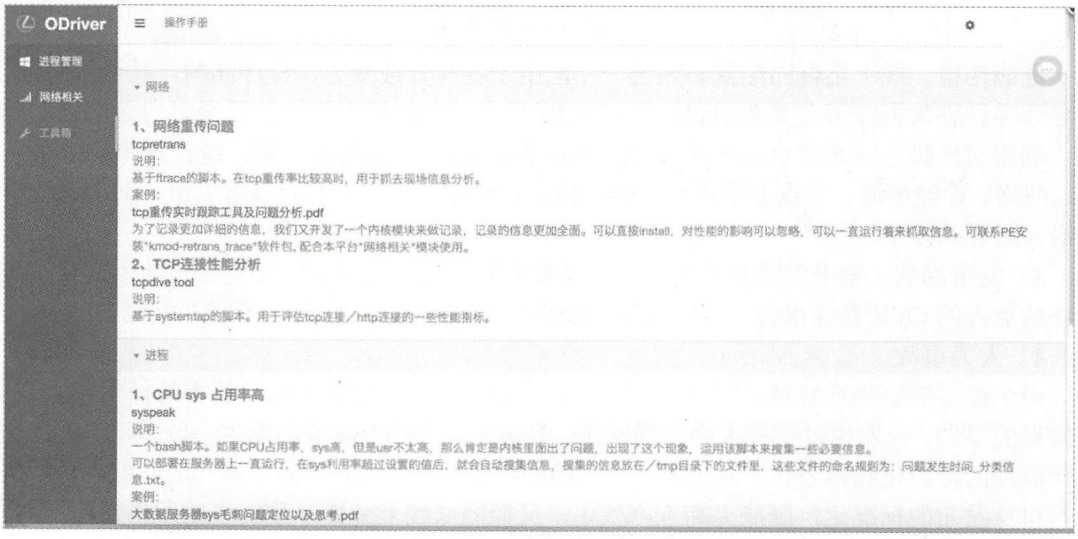


图 3-25　系统问题分析定位平台

3. 成本

降低机器成本和运维成本是一项长期而艰苦的工作。在业务高速发展的过程中，很可能会

忽视机器成本给公司运营带来的压力。一旦开始降低成本，会发现有很多降低的空间，在这个过程中，我们也总结了几条降低成本的最佳实践和经验。

1）资源池化。各个项目组的开发人员往往习惯为各自的项目预留一定的缓冲空间。这些缓冲的机器池往往会变成业务方的自留地。将业务接入 PaaS 平台的一大好处是通过容器的动态申请和调度，把小池子逐个消除，形成统一的资源池。

2）混部和调度。提升容器部署密度以后，对业务带来的风险是硬件故障对业务稳定性的影响会比之前更高。根据业务基线属性的不同，将资源消耗不同类型的业务混部在一台宿主机上，可以很好地提升单机的资源利用率。

3）弹性能力。电商的行业特征决定了每年都会出现 618、双 11、双 12 等大促的峰值流量。因此，需要购买大量的机器应对这些峰值，峰值压力过后，机器资源有很大的闲置和浪费。因此，在大促期间动态地租用部分公有云资源，在平时只需保留满足日常流量的机器池即可。

3.2　“自我突破”：关键技术方案和创新点

下面介绍美联容器云平台的技术方案，特别是关键的技术方案和创新点。

3.2.1　版本演进

美联容器云平台的很多组件都经历了两次或两次以上的版本演进，主要目的是增强隔离性、稳定性和性能，如图 3-26 所示。

图 3-26　美联容器云平台架构演进

1. 内核版本

对于容器来说，内核的选型是非常关键的，因为大量容器的隔离性和稳定性的提升都依赖内核。

美联的内核从 CentOS6 的 2.6.32，演进到 CentOS7 自带的 v3.10，又演进到了 4.4.95。之所以选择 4.4.95 版本，是因为它对 OverlayFS 的支持更加完善，支持 pid_max 的隔离，也是社区长期支持的稳定版本。

2. Docker 版本

美联的 Docker 版本从最早的 v1.3.2 升级到了 v1.10.3，又演进到了 v1.13.1。因为从 v1.12 开始，Docker 社区实现了 Docker daemon 和 Containerd 的分离，当升级 Docker daemon 时才能够做到真正的业务无感知。

3. 存储后端

美联的存储后端也从 CentOS 7 默认的 Devicemapper，演进到了社区推荐的 OverlayFS。

Devicemapper 的缺陷主要在于随机 I/O 很容易被放大，从而导致业务的 I/O 性能下降。Devicemapper 默认的块大小是 512KB，而 ext4 的默认大小只有 4KB。即使运行在 Devicemapper 上的容器只写 1 个字节，实际也会写入 512KB 个字节。一个普通的随机写，实际上是被放大了 512/4 = 128 倍，这对于大量随机写的业务来说是非常致命的。 曾经遇到过的一个真实案例是，一个重 I/O 的业务容器化后下降了 50%，通过将 Devicemapper 替换成 OverlayFS，顺利解决了性能下降的问题，性能几乎与物理机接近。

另一点是运行在 Devicemapper 上的容器容易出现磁盘超配的情况。Devicemapper 是 thin-provision，分配的是虚拟的磁盘空间，因此它并不感知实际物理磁盘的使用情况，一旦出现物理磁盘写满的情况，对业务的影响是致命的。

从 Docker v1.13.1 开始，社区推荐的存储后端，已经从 Devicemapper 变成了 OverlayFS，这也是目前生产环境的默认配置。

3.2.2 关键技术和创新点

1. 隔离性

众所周知，容器的隔离性弱于 KVM / Xen 这样的全虚拟化技术。在 Kubernetes 层面，虽然实现了 CPU 和内存的隔离，但是针对磁盘 I/O 和网络 I/O 的隔离是缺失的。在实际生产环境中，这两者的隔离性显得尤为重要。

1）异步 I/O 隔离性

什么是异步 I/O

如图 3-27 所示，在 Linux 中写磁盘通常有三种方式。

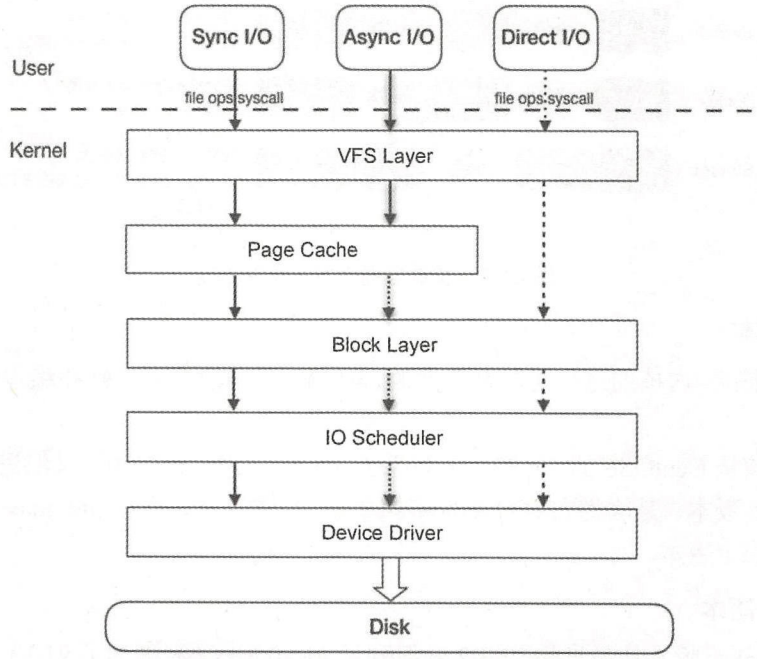


图 3-27　Linux 中写磁盘的三种方式

➢ Direct I/O。不使用 Page Cache，直接写入块设备，通常这种方式都是由应用维护自己

的 Cache 而不去使用内核的 Page Cache。数据会直到被写入到设备驱动的队列里，写操作才返回。如右侧的虚线所示，中间会经过 Block Layer。

- 同步写。这种方式如左侧的实线所示，会经过内核的各个层，一直写入设备驱动的队列里面才结束，同样会经过 Block Layer。
- 异步写。即 Buffer I/O 方式。这种方式如中间的实线所示，它只写入 Page Cache 就结束了。没有到达 Block Layer。百度百科的解释：

> 用于回写功能的部分缓存称为写缓存（Write Cache）。在一套写缓存打开的存储中，操作系统发出的一系列写 I/O 命令并不会被挨个执行，这些写 I/O 的命令会先写入缓存中，然后再一次性将缓存中的修改推到磁盘中，相当于将相同的多个 I/O 合并成一个，多个连续操作的小 I/O 合并成一个大的 I/O，还有就是将多个随机的写 I/O 变成一组连续的写 I/O，这样能减少磁盘寻址等操作消耗的时间，大大提高磁盘写入的效率。

为了提高 I/O 性能，80%以上的应用写都采用异步写形式。对于这种方式而言，应用写到 Page Cache 就返回了。而数据从 Page Cache 写入到 Device Driver，则是由 Writeback 实现的，如带阴影的虚线所示。Writeback 是在文件系统里的一个内核线程，各个文件系统的 Writeback 有一定区别，例如 Ext4 叫 Flush，XFS 叫 Writeback。

数据从设备驱动到真正的落盘（Backing-Device），是通过中断上下文实现的，即设备驱动里的数据就绪后会触发一个中断给控制器，然后控制开始做数据的搬移。

Linux 系统现在能够针对进程的 I/O 做限制（I/O throttle）的地方是在 Block Layer，比如 Cgroup 的子系统 Blkcg 就是在 Block Layer 做的 I/O throttle。

通过图 3-27 可以看到，只有 Direct I/O 和 Sync I/O 这两种方式会经过 Block Layer，而 Async I/O（即 Buffer I/O）这种方式没有经过 Block Layer，所以没有办法针对进程进行 Buffer I/O 的控制。所有进程的 Buffer I/O 都是统一由 Writeback 线程来写入到 Backing-Device（比如磁盘），Writeback 线程并不清楚要回写的脏页是哪一个进程产生的，这就导致了 Buffer I/O 的失控。

换句话说，在 Linux kernel 4.2 之前的 Cgroup v1，默认只支持对同步 I/O 的限速，对异步 I/O 还无法做到真正的限速。

因此，在 CentOS 7 默认的 3.10 内核上，容器的 I/O 隔离性会有比较大的问题。

一刀切的做法

所谓一刀切的做法，就是直接控制 Writeback 线程的限速行为。反正所有的 Buffer I/O 都由它来回写，那不管三七二十一通过它来控制所有的 Buffer I/O 好了。这也是现在业界最通用的做法（其实也是无奈的选择）。

Linux 内核提供了一些接口来微调 Writeback 线程的行为，如下几个参数：

```
$ sysctl -a | grep dirty
vm.dirty_background_ratio = 10
vm.dirty_background_bytes = 0
vm.dirty_ratio = 40
vm.dirty_bytes = 0
vm.dirty_writeback_centisecs = 500
vm.dirty_expire_centisecs = 3000
```

通过这种一刀切的控制方式，在某些场景下确实可以维持 I/O 的稳定性。来看一个成功案例：

某个业务人员在做性能压测时，相同配置的机器，有些机器的 QPS 可以达到 2500，有一些机器的 QPS 却只有 1800。经过分析发现，这些 QPS 较低的机器是因为日志量过大导致 I/O 不稳定，从而影响到业务进程。然后在这些 QPS 低的机器上调整了上文提到的几个参数，使得 Buffer I/O 的行为更加的均匀，不再突发，最终这些 QPS 低的机器也能够达到 2500 了。

之所以可以这么做，是因为这个业务线程对于 I/O 不敏感，只有日志线程存在大量的 I/O，而日志线程又不是那么的重要。所以，一刀切的主要目的是限制日志线程，让日志线程的 I/O 不再 burst，而变成 drain。

然后问题就来了：如果业务线程本身对 I/O 比较敏感，I/O 甚至是性能瓶颈，那么一刀切的这种处理方式无疑会给业务雪上加霜。

这种方式的缺陷

减少 I/O 的 burst 的关键是让系统里面的脏页面尽量少一些，这样就不会有大量的突发 I/O。主要的方式，一是让 Writeback 工作的勤奋些，不停地去刷脏页面（通过 vm.dirty_background_ratio 来控制），二是从源头上减少脏页面，强制让异步写变成同步写（通过 vm.dirty_ratio 来控制）。

更加优雅的解决方案，当然是能够控制每个线程的 Buffer I/O，即 I/O QoS。根据不同进程 I/O 的重要性，来分别控制 I/O。于是，内核社区诞生了 Cgroup v2。

Cgroup v2 相对于 Cgroup v1 已经不仅仅是改造，而是完全重写了一套，跟 Cgroup v1 并不兼容。在 kernel-4.2 版本以后，Cgroup v1 和 Cgroup v2 都处在共存的状态，随着 Cgroup v2 的不断完善，相信 Cgroup v1 慢慢会退出历史舞台。

这里对比一下 Cgroup v1 和 Cgroup v2 的 Writeback 逻辑架构：

Cgroup V1 时的 Writeback 如图 3-28 所示。

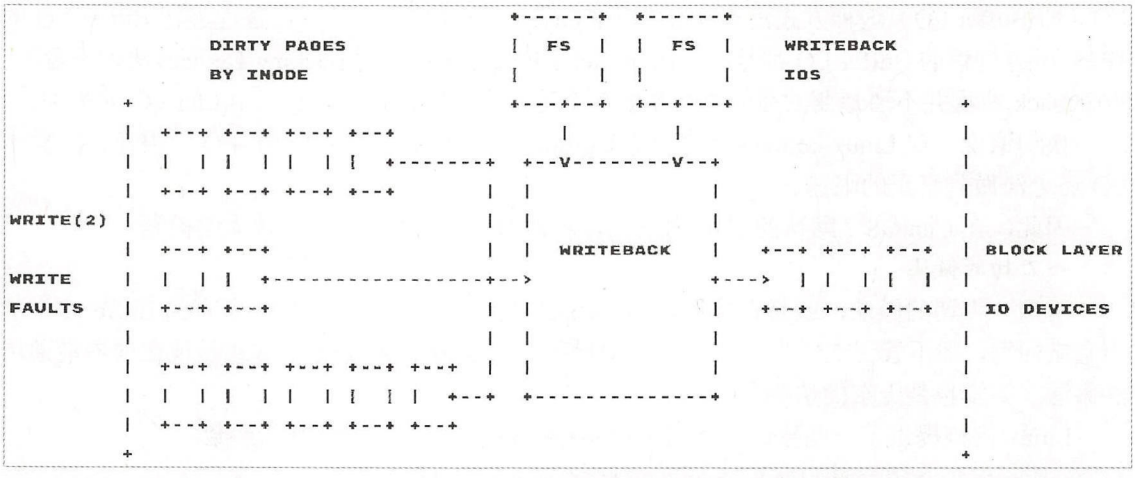


图 3-28　Cgroup v1 时的 Writeback

Cgroup v2 时的 Writeback 如图 3-29 所示。

Writeback 不再是一个整体，而是有了很多的 Group 来控制。借助 Cgroup v2 的特性，就可以实现对所有 I/O 的限速了。

但目前 Cgroup v2 在内核中还不够成熟，而且只支持 ext4，还不支持 xfs，因此我们参考了 Cgroup v2 的思路，自研实现了异步 I/O 隔离的方案。

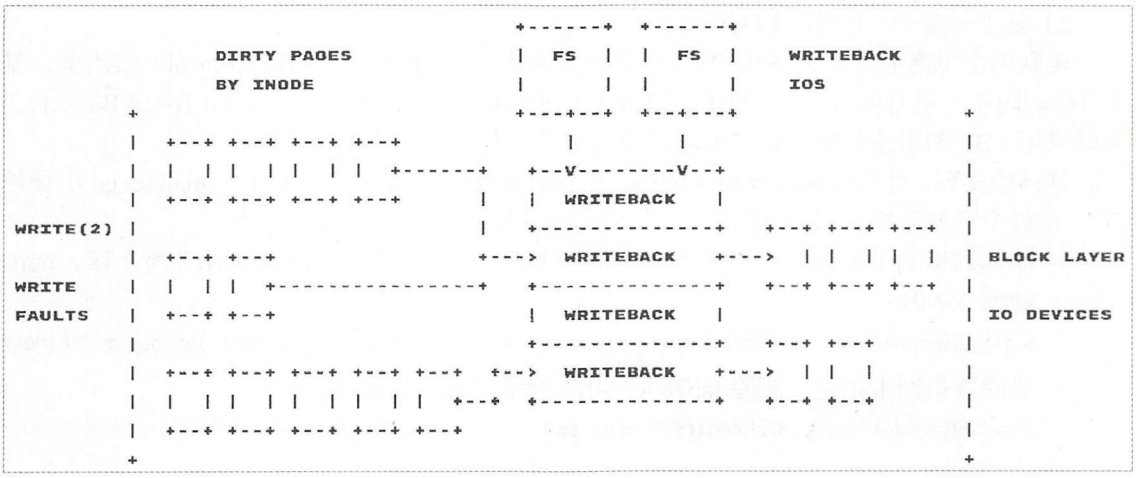


图 3-29　Cgroup v2 时的 Writeback

美联的做法

在 Docker 容器里，其实每个容器都会存在一个相应的 blkcg 和 memcg 的 Cgroup，限制内存和磁盘 I/O 资源的使用。但是在原生的 Cgroup v1 的实现当中，memcg 和 blkcg 是完全独立的子系统并且彼此之间没有关联关系，这样就导致在 Writeback 线程对脏数据落盘时只能找到 memcg 信息，而找不到 blkcg 相关的信息。

然而，Cgroup 的 I/O throttle 只能通过 blkcg 的信息在 block 层进行 I/O 限速。所以，充分考虑到 Cgroup v1 和 Cgroup v2 之间接口的不兼容性带来的影响以及实现的健壮性，美联的方案是通过对容器的 memcg 和 blkcg 之间建立一种映射，并且在 Writeback 线程回刷脏数据时，通过 memcg 和这种映射关系找出 blkcg 相关的信息，传递给块层来达成限速的目的。

实例验证

对/dev/sdb(8:16)设备进行测试。当没有建立 blkcg 和 memcg 的映射关系时，对异步 I/O 是没有限制效果的。

```
   mkdir /sys/fs/cgroup/blkio/cgroup1
   echo "8:16 1048576" > /sys/fs/cgroup/blkio/cgroup1/blkio.throttle.write_bps_device
   mkdir /sys/fs/cgroup/memory/cgroup1
   echo $$ > /sys/fs/cgroup/blkio/cgroup1 /tasks
   echo $$ > /sys/fs/cgroup/memory/cgroup1/tasks

   fio -directory=/mnt/sdb/cgroup1 -direct=0 -iodepth 8 -thread -rw=randwrite -ioengine=psync -bs=4k
-size=10G -numjobs=1 -runtime=60 -group_reporting -name=mytest
```

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.03    0.00    0.09    3.16    0.00   96.71

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sdb               0.00     0.00    0.00  565.00     0.00   141.25   512.00   144.56  264.90    0.00  264.90   1.77 100.00
```

当建立 blkcg 和 memcg 的映射关系时，对异步 I/O 有限制效果。

```
   cat /sys/fs/cgroup/blkio/cgroup1/blkio.id > /sys/fs/cgroup/memory/cgroup1 /memory.ass_id
   fio -directory=/mnt/sdb/cgroup1 -direct=0 -iodepth 8 -thread -rw=randwrite -ioengine=psync -bs=4k
-size=10G -numjobs=1 -runtime=60 -group_reporting -name=mytest
```

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.13    0.00    0.47    0.69    0.00   98.72

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sdb               0.00     0.00    0.00  125.00     0.00     1.00    16.45     0.00    0.00    0.00    0.00   0.00   0.00
```

2）基于开源 TC 的网络 I/O 隔离

美联网络隔离的方案，是在 OVS 的基础上利用开源的 TC（Traffic Control）实现的。基于 TC 的限速一般有两种类型：TBF（Token Based Filter）和 HTB（Hierarchical Token Bucket）。美联采用 TBF 的限速策略，已经能满足绝大多数业务场景的需求。

具体的实现，是修改 Kubernetes 源码，在 pkg/kubelet/dockertools/docker_manger.go 中调用 TC，在每个容器连接的 tap 设备上设置相应的限速值。

- 添加虚拟网卡限速。在 tap7ebccd4ff47 网络设备上，设置 2GB 的网络带宽上限，burst 速率 25MB/s。

```
$ tc qdisc replace dev tap7ebccd4ff47 root tbf rate 2000000000 latency 5ms burst 25000000
```

- 删除虚拟网卡限速。删除 tap7ebccd4ff4 网络设备上的限速。

```
$ tc qdisc del dev tap7ebccd4ff47 root tbf
```

2. 高可用

1）Kubernetes 自带的 Pod 驱逐策略

Kubernetes 有一个特色功能叫 Pod eviction，它在某些场景下如节点 NotReady 资源不足时，把 Pod 驱逐至其他节点。从发起模块的角度，Pod eviction 可以分为两类。

- Kube-controller-manager：周期性检查所有节点状态，当节点处于 NotReady 状态超过一段时间后，驱逐该节点上所有 Pod。
- Kubelet：周期性检查本节点资源，当资源不足时，按照优先级驱逐部分 Pod。

Node controller 的 Pod 驱逐策略

Kube-controller-manager 周期性检查节点状态，每当节点状态为 NotReady，并且超出 podEvictionTimeout 时间后，就把该节点上的 Pod 全部驱逐到其他节点，其中具体驱逐速度还受驱逐速度参数、集群大小等的影响。最常用的 2 个参数如下。

- - pod-eviction-timeout：NotReady 状态节点超过该时间后，执行驱逐，默认为 5 min。
- - node-eviction-rate：驱逐速度默认为 0.1 pod/s。

当某个 zone 故障节点的数目超过一定阈值时，采用二级驱逐速度进行驱逐。

- - large-cluster-size-threshold：判断集群是否为大集群，默认为 50，即 50 个节点以上的集群为大集群。
- - unhealthy-zone-threshold：故障节点数比例，默认为 55%。
- - secondary-node-eviction-rate：当大集群的故障节点超过 55% 时，采用二级驱逐速率，默认为 0.01 pod / s。当小集群故障节点超过 55% 时，驱逐速率为 0 pod / s。

Kubelet 的 Pod 驱逐策略

Kubelet 周期性地检查本节点的内存和磁盘资源，当可用资源低于阈值时，则按照优先级驱逐 Pod，具体检查的资源如下：

- memory.available。
- nodefs.available。
- nodefs.inodesFree。
- imagefs.available。
- imagefs.inodesFree。

以内存资源为例，当内存资源低于阈值时，驱逐的优先级大体为 BestEffort > Burstable > Guaranteed，具体的顺序可能因实际使用量有所调整。当发生驱逐时，Kubelet 支持 soft 和

hard 两种模式，soft 模式表示缓期一段时间后驱逐，hard 模式表示立刻驱逐。

2）落地思考

对于 Kubelet 发起的驱逐，往往是资源不足导致，它优先驱逐 BestEffort 类型的容器，这些容器多为离线批处理类业务，对可靠性要求低。驱逐后释放资源，减缓节点压力，弃卒保帅，保护了该节点的其他容器。无论设计方面，还是实际使用情况方面，该特性都非常好。

对于由 kube-controller-manager 发起的驱逐，效果需要商榷。在正常情况下，计算节点周期给 master 上报心跳，如果心跳超时，则认为计算节点 NotReady，当 NotReady 状态达到一定时间后，kube-controller-manager 发起驱逐。然而，造成心跳超时的场景非常多，例如：

- 原生 Bug：Kubelet 进程彻底阻塞。
- 误操作：误把 Kubelet 停止。
- 基础设施异常：如交换机故障演练、NTP 异常、DNS 异常。
- 节点故障：硬件损坏、掉电等。

从实际情况来看，真正因计算节点故障造成心跳超时的概率很低，反而由原生 Bug、基础设施异常造成心跳超时的概率更大，造成不必要的驱逐。

在理想情况下，驱逐对无状态且设计良好的业务方影响很小。但是并非所有的业务方都是无状态的，也并非所有的业务方都针对 Kubernetes 优化其业务逻辑。例如，对于有状态的业务，如果没有共享存储，异地重建后的 Pod 完全丢失原有数据；对于关心 IP 层的业务，异地重建后的 Pod IP 往往会变化，虽然部分业务方可以利用 Service 和 DNS 来解决问题，但是引入了额外的模块和复杂性。

除非满足如下需求，不然请尽量关闭 kube-controller-manager 的驱逐功能，即把驱逐的超时时间设置得非常长，同时把一级和二级驱逐速度设置为 0。否则，实际效果弊大于利。

- 业务方要用正确的方式使用容器，如数据与逻辑分离、无状态化、增强对异常处理等。
- 分布式存储。
- 可靠的 Service、DNS 服务或者保持异地重建后的 IP 不变。

3）落地措施

为了避免因交换机故障或者基础设施等异常造成不必要的大规模驱逐，蘑菇街把 large-cluster-size-threshold 值调大（默认为 50），可以让系统保持在小集群状态，并且把 pod-eviction-timeout 适当调大(默认为 5min)，可以避免短时网络中断引发不期望的 Pod Eviction。

另外，美联还修改了 Kubernetes 源码，在 Pod 上增加了 preserve=true 的标记位，针对 Redis 一类的有状态业务，无论网络中断，还是机器重启，始终不进行迁移。

3. 网络

Kubernetes 自身并不包含网络模块，业界常见的网络方案有 Flannel/Calico 等。美联的容器云从 OpenStack+Docker 演进而来，同时将 OpenStack 的网络方案移植到了 Kubernetes 上来。

1）基于 OVS 的 mogunet 网络插件

美联自研的这一套基于 OVS 的网络插件叫 mogunet，与 OpenStack 社区推广的 Kuryr 项目（https://github.com/openstack/kuryr）很类似，如图 3-30 所示。通过 OpenStack Neutron 和 OpenvSwitch 管理容器的网络资源。

这样做的好处有很多。首先，Neutron 可以统一管理虚拟机、物理机和容器的网络资源分配；同时因为采用的是 Vlan，它的隔离性和性能开销也能得到保证。

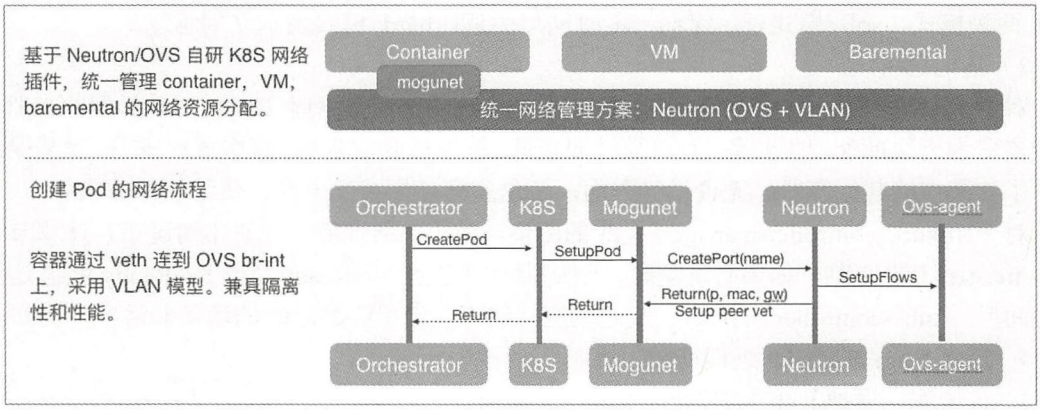


图 3-30　mogunet 网络插件

未来，美联也可以使用 Neutron ml2 里已有的网络插件，来实现更高的隔离性和扩展性，比如实现防止 ARP 广播风暴等高级特性（L2 population and ARP proxy）。如图 3-31 所示为基于 OVS 的容器网络示意图，同一台宿主机上的容器可以分布在不同的 C 类段里，并且和宿主机的网段分开，通过 Vlan 连接到 OVS 的网桥上，最后通过 eth0 / eth1 连接到 TOR 交换机上。这和 KVM 虚拟机采用的网络模式是一样的。

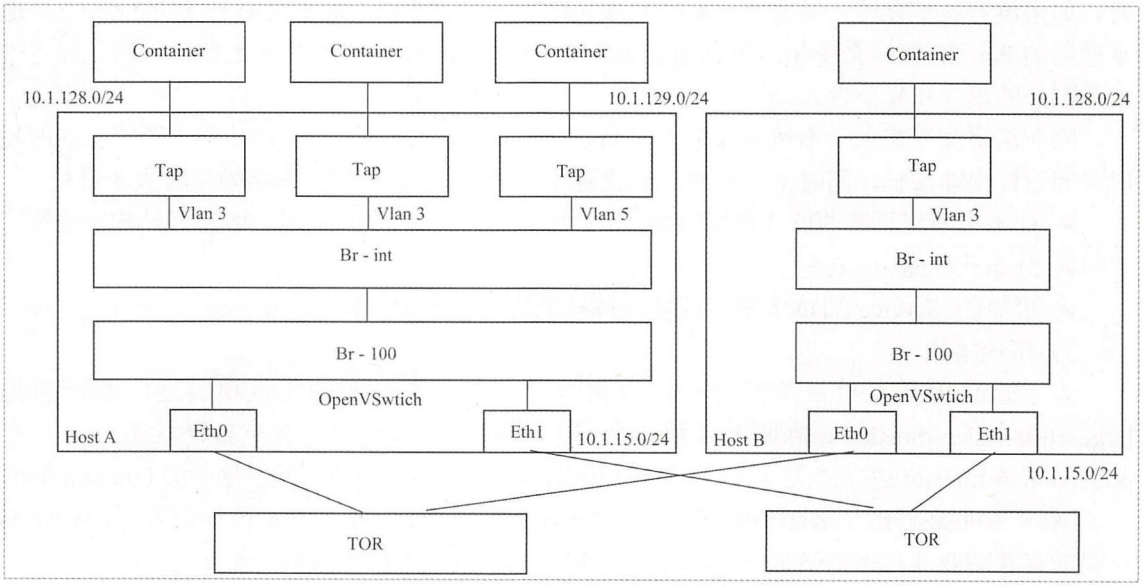


图 3-31　基于 OVS 的容器网络示意图

2）容器跨主机迁移时保持 IP 不变

这个需求来自真正的业务方。在实际的业务场景中，IP 往往与 DNS 服务器、Client 的配置、服务中心等第三方服务强关联。因此，业务方希望容器发生 Pod 跨主机迁移时能保持不变。

在 Kubernetes 中通过 Service 的形式，将后端的 Endpoint IP 隐藏起来，但在实际业务场景中，还有部分业务需要通过 TCP 直连后端的服务。

利用 Kubernetes 的 Statefulset Pod Name 在迁移时保持不变的特性，我们将 Pod 的名字和 Namespace 映射成 Neutron 中的一个 port，每个端口会对应一个唯一的 IP 地址，这样就可以通

过 Pod Name 找回原来分配出去的 port，从而找回原来的 IP 地址。

举个例子，在一开始创建容器时，会先将 port 和 IP 信息记录到 Neutron 里面去，如图 3-32 所示。假设这时的源节点发生故障了，controller manager 会把 StatefulSet 调度到其他的 Node 上，在 Kubelet 启动容器时，网络插件会根据 Pod 和 Namespace 对 Neutron 查询 Port 是否存在，如果存在则重用这个 IP 地址，并且将 device owner 更新成自己。这样就实现了容器跨物理节点迁移时 IP 不变的特性。

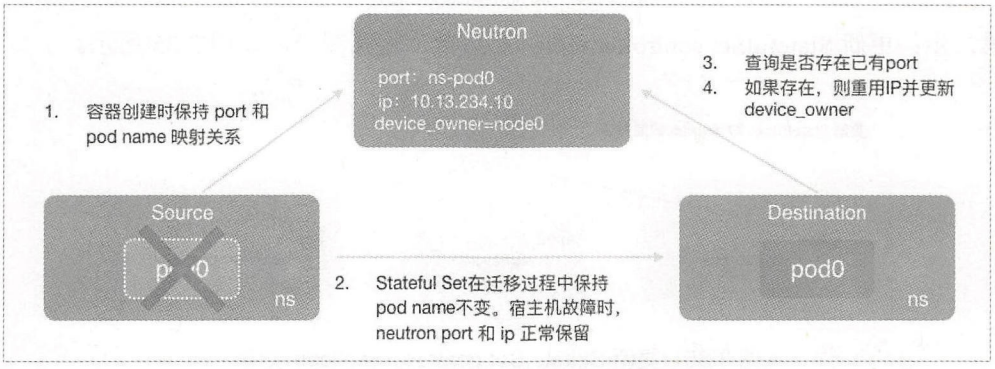


图 3-32　容器跨主机迁移时保持 IP 不变

3）指定 IP 删除 Pod

对于有状态业务，使用 StatefulSet 时会遇到 StatefulSet 自身的一个缺陷。就是 StatefulSet 只支持按序删除 Pod。实际上 StatefulSet 只支持扩容和缩容接口。如果 Pod 创建的顺序是 Pod1、Pod2、Pod3，那么 StatefulSet 删除 Pod 的顺序就是 Pod3、Pod2、Pod1。

StatefulSet 的扩缩容逻辑并不能满足内部一些有状态业务的实际使用场景，比如某个实例出现业务逻辑问题时，需要下线销毁，或者宿主机需要回收而必须下线某个实例，并且要求减少一个 Replicas 数量，如图 3-33 所示。

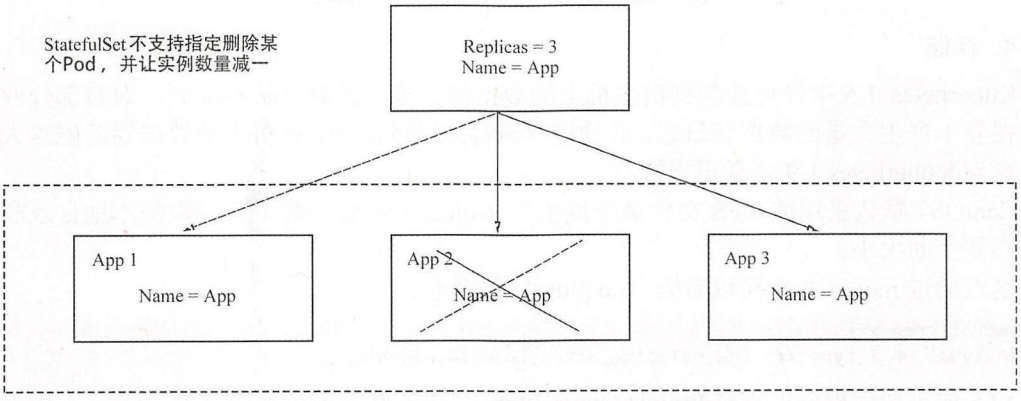


图 3-33　StatefulSet 不支持直接删除中间的 Pod

因此，通过优化 StatefulSet 缩容的顺序，实现了指定 IP 删除容器的特性。具体来说，调整 StatefulSet Controller 缩容的排序，对于 label 为 to-be-delete=1 的 Pod，将其删除顺序调整至最高。这需要以下三个步骤来完成删除的动作。

第一步：在准备删除的容器上添加一个 to-be-delete=1 的 label，如图 3-34 所示。

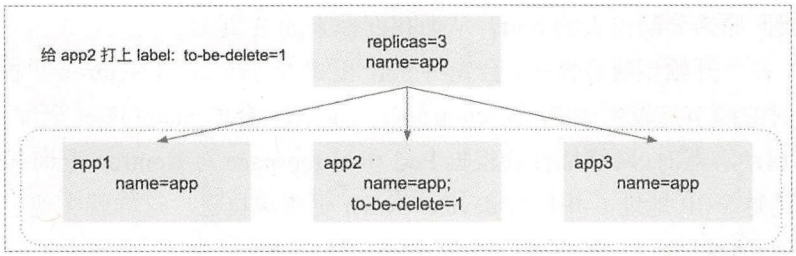


图 3-34　添加 to-be-delete=1

第二步：更新 StatefulSet controller replicas 数量，将其减一，如图 3-35 所示。

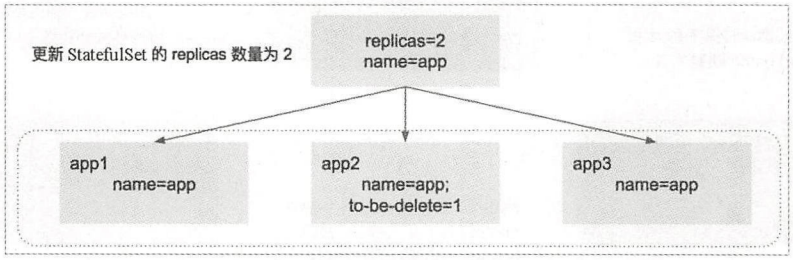


图 3-35　更新 StatefulSet controller replicas 数量

第三步：通过 Kubernetes 删除 Pod 的接口，将 App-2 的 Pod 删除，如图 3-36 所示。

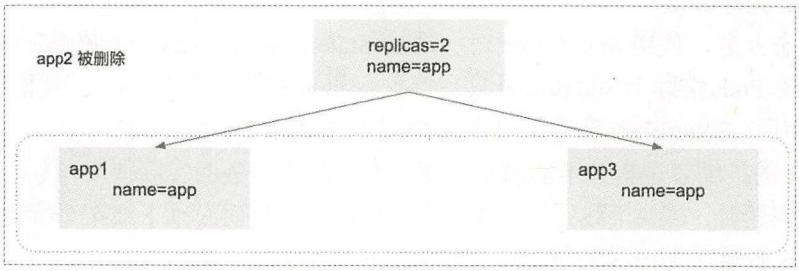


图 3-36　通过 Kubernetes 删除 App-2

4. 存储

Kubernetes 1.5 中针对挂载到宿主机上的数据卷并没有做磁盘配额限制，而有部分业务会在数据卷上产生大量的数据和日志，必须进行磁盘配额的限制。针对本地数据卷的磁盘大小隔离，直到 Kubernetes 1.9 才真正支持。

CentOS7 默认采用的 XFS 文件系统提供了 Project Quota 功能，可以限制本地目录和子目录的存储空间大小。

这点通过 mount 命令可以看到‘noquota’关键字。

```
$ mount|grep xfs
/dev/vda1 on / type xfs (rw,relatime,attr2,inode64,noquota)
```

XFS 如果要使根分区支持 Project Quota 功能，则需要修改内核系统参数，在/etc/default/grub 中的 GRUB_CMDLINE_LINUX 最后添加“rootflags=uquota,pquota”。比如：

```
GRUB_CMDLINE_LINUX="clocksource_failover=acpi_pm crashkernel=auto rhgb quiet rootflags=uquota,
pquota"
```

将原来的/boot/grub2/grub.cfg 备份后，通过 grub2-mkconfig 生成一个新的 grub.cfg，重启机器以使之生效。

```
cp /boot/grub2/grub.cfg /boot/grub2/grub.cfg.orig
```

```
grub2-mkconfig -o /boot/grub2/grub.cfg
reboot
```

重启后先验证一下 XFS 的 Project Quota 能力生效了。

```
$ mount|grep xfs
/dev/vda1 on / type xfs (rw,relatime,attr2,inode64,usrquota,prjquota)
```

然后可以在容器分区上通过 xfs_quota 来设置磁盘配额大小。

```
xfs_quota -x -c 'limit -g bsoft=9g bhard=10g cent' /docker/container-id/volume
```

可以通过以下命令来查看磁盘配额和实际的使用情况

```
xfs_quota -x -c 'report -h -g' /docker/container-id/volume
```

我们在 Kubernetes 中实现了一个新的 volume plugin 来实现 XFS 的配额能力。具体做法是：

- 在/pkg/api/v1/types.go 增加一个 volume source 类型.
- /pkg/api/types.go 增加一个相同的数据类型。
- /pkg/volume 目录新增一个子目录比如 iso_xfs_dir。
- 实现 VolumePlugin 接口，可以参考已有实现。
- 在 kubelet 代码中注册 Plugin。
- make update 生成 auto-generated files。

5. 镜像管理

1）镜像分层方案

镜像提供了应用的运行环境，镜像管理会影响应用管理的效率和镜像中心的存储空间。以应用的管理效率优先，兼顾镜像中心的存储空间。镜像由 Dockerfile 制作生成，分层管理，如图 3-37 所示。通过合理的分层来提高部署效率。将通用性强、公用广泛的组件放到下层，将经常变化的组件放到上层。下层是容器间共享的，更好的复用下层是提升部署效率的关键。另外，由于 Docker 镜像中心的技术实现，下载每层镜像都会进行 MD5 校验，因此镜像层数不宜过多，否则多次 MD5 校验会影响下载镜像的效率。

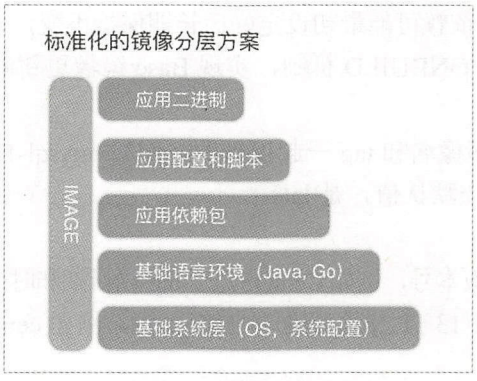


图 3-37　镜像分层方案

从逻辑管理上将镜像分为 5 层。逻辑上的分层并不一定和 Docker 底层镜像的分层严格对应，Docker 底层镜像分层可能会多于 5 层。但是二者分层管理的趋势和策略一致。

基础系统层

逻辑上又可以细分为两层。目前只提供 CentOS7 的系统，最下层为 CentOS 7.2 自带的系统包，上面是公司需要的基础运维工具，例如 ldap、curl、tsar，以及 DNS 配置、内部账户等。

这层镜像只有 1 个，被所有的应用共享（所有的应用镜像都会用到它）；不和其他的运维系统有交互，由虚拟化负责提供并维护，一般尽量不做镜像变更。

基础语言环境

编程语言依赖的基础镜像层，每种语言有不同的依赖环境，例如 Java、C++、PHP、Go 等。每种语言也可能对应不同的依赖，例如 Java JDK 1.7、Java JDK 1.8。

应用按照编程语言分类，找到对应的 language_base 层的依赖。该层会有多个镜像，前期可能只有几个，后期可能会有十几个甚至更多，但即便更多，也能够维护，工作量并不大。

这层镜像一般也不会有修改，但会有新增，例如新增一种语言依赖环境。

应用依赖包

应用的特殊第三方依赖，以及公司内部的常用依赖组件，例如监控 Agent、运维 Agent、安全 Agent 等。

每个应用的第三方依赖可能会不同，因此每个应用（分组）都对应自己的一层。同一个应用分组，这层镜像只保留一个。

一旦应用（分组）的依赖变更、修改了配置，会重新构建这层镜像，并把原来的镜像删除。

应用配置和脚本

应用程序依赖组件的配置文件和脚本等。同前面说的第三方依赖一样，每个应用（分组）都对应自己的一层，并且只保留一个。一旦有配置变更，会重新构建这层镜像，并把原来的镜像删除。

应用二进制

最上层就是应用程序本身了。将可运行的应用程序的文件打入到镜像。保留最近的 3 个镜像（一个镜像对应一次发布），更早的镜像需要删除。

另外，基础 Base 镜像的组件更新也会体现在这层。Base 镜像一般不会更新或者尽量不去更新，如果有更新的需求，就把更新的内容体现在应用程序层。放在应用程序层而不是放到应用依赖层，主要是考虑到应用依赖可能最初设定好后长期保持不变，不会触发镜像更新。

可以通过 Dockerfile 的 ONBUILD 机制，实现 Base 镜像更新的这个“钩子”。

2）命名规范

Docker 的镜像是通过镜像名和 tag 一起识别的，例如 mysql-1.6:v2 中，mysql-1.6 是镜像名，v2 是 tag。其中 tag 有个默认值，是 latest。

基础系统层

镜像名中带有操作系统版本号，例如 CentOS7.2。tag 以日期和时间命名，例如 201708251328，表示它是 2017 年 8 月 25 日 13 点 28 分制作（上传）的。例如 centos7.2_base:201708251328。

基础语言环境

镜像名包含了语言信息，例如 Java、Go。tag 以语言的依赖包版本号或语言编译器本身的版本号命名区分。示例：java:jdk1.8-tomcat8, go:1.8。如果语言没想好如何通过 tag 区分，就先使用默认的 latest tag，例如 php:latest。tag 里以“-”为字段的风格，包名的版本号用“.”和“_”。

应用依赖和配置脚本

镜像名中包含应用名和应用分组名，以“斜杠/”区分，最后以“_baseline”结尾。tag 号

以日期和时间命名。

示例：public_resource_pool/public_resource_poolhost_baseline:201708251328。

应用二进制

镜像名是应用名和应用分组名，同前面说的第三方依赖层的命名。tag 是代码的 commitID。示例：paas/paas_apiserverhost:5cb54ca379e5fd6df2b057c30a12edd6b6b1bb6e。

3）镜像管理中心

开源的 Docker Registry（后命名为 Docker Distribution）只提供了基本的镜像拉取和推送功能，并不支持镜像搜索、权限控制、高速下载、多机房管理等方面的能力。因此，有必要开发一个适合企业级使用的镜像管理中心，如图 3-38 所示。

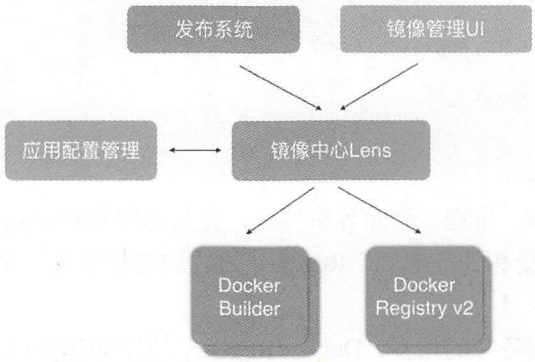


图 3-38　镜像管理中心 Lens

由于美联研发的时间比较早，并没有采用业界比较流行的 Harbor 方案，而是用 Go 语言开发了自研的镜像管理中心，我们叫它 Lens。

镜像管理中心 Lens 是 PaaS 平台很重要的一部分，Lens 主要支持以下功能：与应用配置中心对接，获取应用配置信息，自动生成 Dockerfile；自动构建应用 Docker 镜像，并推送至不同环境的 Docker Registry；提供面向用户的 Docker 镜像管理 UI，提供镜像的快速搜索、详情展示、查询构建日志、删除等功能；支持两个跨机房的 Docker Registry 之间进行镜像同步，比如开发环境的镜像同步至生产环境，支持定时同步；监听 Docker Registry 的拉取或推送事件，对接基于 LDAP 的权限管理模块，对镜像的增删改查进行控制。

➢ 镜像自动构建

得益于应用配置的标准化，通过配置中心可以获取到应用的启停命令、部署目录、依赖软件包环境等信息，因此可以根据配置系统中的信息生成 Dockerfile，并且通过 Lens 发送 build 命令到 Docker daemon 上进行镜像构建，最后将构建好的镜像自动上传到镜像中心。该功能不仅提供了镜像构建的自动化，还对业务方屏蔽了 Dockerfile 的专业知识。

➢ 镜像跨机房同步

要能实现 Docker Registry 之间的同步，必须了解 Docker Registry v2 存储镜像的格式。Registry 使用两个对象记录一个镜像：manifest 用于描述镜像的信息；blob 是镜像的数据块，也就是镜像真正的内容，保存在文件系统中。

镜像在 Registry 之间的同步，实际上是 manifest 和 blob 的同步。镜像同步的步骤如下：获取源端镜像的 manifest；获取源端镜像的 blob 列表；在目的端 Registry 检查是否存在这些 blob，已存在的 blob 不需要同步；从源端同步目的端不存在的 blob；同步 manifest；验证镜像

是否同步成功。

可以使用 Docker Registry 提供的 API 完成以上大部分操作。

➢ Docker Registry 事件监听

Lens 作为镜像管理的组件，也拥有自己的数据库，记录镜像相关的信息。Lens 数据库的信息要与 Registry 存储的镜像数据一致。有些镜像管理操作是从 Lens 发起，最终在 Registry 上完成操作的。也有些镜像管理可能会不经过 Lens，直接操作 Registry，例如手动向 Registry 上推送镜像。为了保证两边数据的一致性，Lens 需要监听 Registry 上镜像的变更。

Registry 支持事件通知，通过配置 notifications 定义事件通知的规则，例如下面的配置。

```
notifications:
  endpoints:
    - name: alistener
      url: http://$IP:$port/registry/event
      headers:
        Authorization: xxxxx
      timeout: 500ms
      threshold: 5
      backoff: 15s
```

当 Registry 发生拉取、推送、删除等事件时，会发送通知到 endpoints 的 URL，这个 URL 是 Lens 提供的，用于接受和处理通知。Registry 的通知支持重发、超时等机制。

➢ 镜像删除

随着时间的推移及业务方的增多，Docker Registry 的空间越来越大，而且因为业务在不断更新，很多老的镜像已经完全没有用了，所以需要把这些镜像删除以节约存储空间。

Registry 通过 manifest 和 blob 记录一个镜像，其自身提供了接口删除 manifest，但需要这些接口可用，我们需要给 Registry 增加以下配置：

```
version: 0.1
log:
  fields:
    service: registry
storage:
    cache:
        layerinfo: inmemory
    filesystem:
        rootdirectory: /var/lib/registry
    delete:
        enabled: true
http:
    addr: :5000delete 选项必须 enabled，否则调用接口会返回 unsupported 的报错
```

另外，美联使用了 Docker Auth 给 Docker 的接口调用进行了鉴权，所以在删除镜像之前，查询镜像概况、获取 manifest、真正调用删除都需要取得不同的动作，然后才能真正调用接口，否则返回鉴权会失败。

删除 manifest 之后，真正的存储空间并没有得到真正回收，还需要调用 Registry garbage-collect 真正执行存储空间的回收。

Docker Registry 的 garbage-collect 使用的是“标记-清理”法：第一步，标记，Registry 扫描元数据，元数据能够索引到的 blob 标记为“不能删除”；第二步，清理，Registry 扫描所有 blobs，如果该 blob 没有被标记，则将其删除。

6. 服务发现

服务发现与健康检查架构如图 3-39 所示。

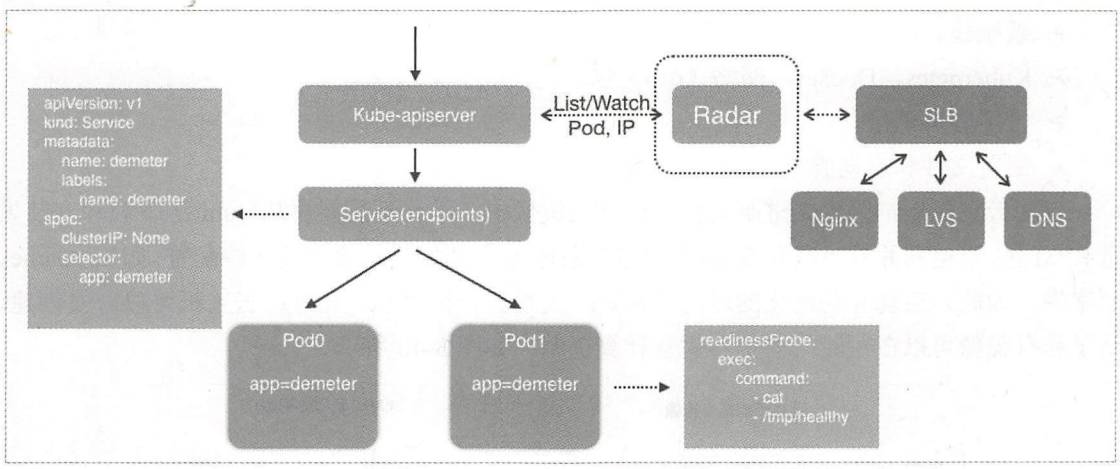


图 3-39　服务发现架构

1）服务发现

目前美联的标准化应用都接入了统一的服务治理中心，这是在 Kubernetes 的 Service 出现前就已经诞生的中间件，因此应用与应用间的访问并没有使用 Kubernetes 原生的 Service 来实现；配置中心在接入业务时，创建出与业务对应的 Service，服务发现组件 Radar 的 informer 会监听业务的 Service，并将 Service 后端健康的 endpoints 注册进 SLB；配置中心还提供 enable/disable 对应 endpoints 的功能，以增强应用对自身应用上下线的能力。

2）健康检查

Kubernetes 原生提供 readiness/liveness probe 的能力，为保证业务的可靠性，美联使用 readiness probe，提供了针对业务四层（TCP）及七层（HTTP）的健康检查规则，用户可以针对自身的需求，指定检测的端口的可用性，来保证对应 Pod 的 IP 是否可以健康地注册进对应的 Kubernetes Service 内，继而被 Radar 发现并注册进 SLB。

7. 弹性能力

2018 年 1 月，美联旗下所有业务迁移至腾讯云黑石机房，租用腾讯云物理机。美联在黑石机房维持一个基础资源池，应对日常流量。在大促场景下，按需租用腾讯公有云云主机资源，将业务水平扩容到公有云环境。

公有云按需收费，弹性能力强。为了最大化节约成本，势必要求快速地从腾讯公有云获取海量资源并完成节点初始化，快速完成业务的部署，这要求：

- 快速从腾讯云获取较多资源的能力。
- 快速初始化多个 Kubernetes 计算节点能力。
- PaaS 平台支持业务方快速伸缩业务的能力，包括高并发创建（下载镜像）容器。

通过制作 Kubernetes 计算节点私有镜像，将 Kubelet 等 RPM 包和初始化脚本注入私有镜像，并把镜像传到腾讯公有云上，基于镜像扩容节点。此外，还打通整个扩缩容流程，使之流程化、自动化，最终实现一键式扩缩容。

1）制作 PaaS 私有镜像

受限于黑石机房和公有云机房之间的专线带宽，同时为了加速 PaaS 计算节点的初始化。制作 PaaS 私有镜像时，把如下模块做进镜像：

- 系统包。
- Kubernetes，Docker，网络（ovs）包。
- 常用容器镜像。
- 配置文件和启动脚本。

启动云主机后，因为二进制文件均已注入镜像中，因此不需要再从 yum 源下载和安装大量 RPM 包，避免高并发下载和安装下的性能瓶颈。整个初始化只要修改一些配置，如 hostname、网络等。因此，启动和初始化的时间非常短，大约在分钟级别。此外，云主机可以并发创建，基于私有镜像可以在短时间内扩容大量计算资源，如图 3-40 所示。

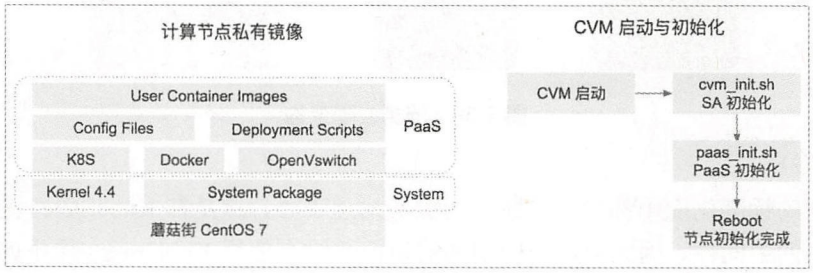


图 3-40　制作 PaaS 私有镜像

2）PaaS 资源池扩缩容

如图 3-41 所示，黑石机房和公有云机房各部署一套完整的 Kubernetes，在日常情况下，公有云机房只保留 Kubernetes master 节点和 etcd 节点，计算节点数为零。大促前，根据压测和水位评估等系统，结合业务方实际需求，预估所需扩容的 IT 资源，在 Portal 上一键式实现扩容。Portal 通过运维系统调用腾讯云 API，完成云主机的创建，并注册到运维系统。云主机启动后完成初始化，注册到 Kubernetes 计算节点。

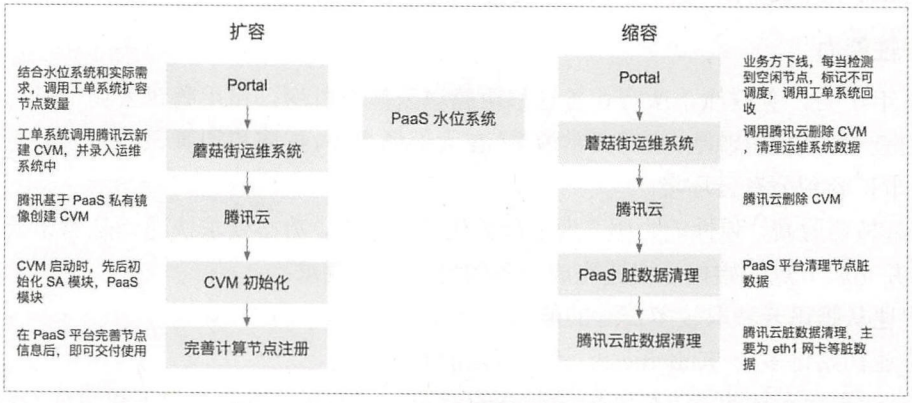


图 3-41　PaaS 资源池扩缩容

当大促结束后，需要回收这些闲置节点。在回收过程中需要注意检查节点是否空闲，为了避免影响业务，先下线空闲节点，下线后还需清理一些脏数据。

3）业务方扩容

公有云机房 Kubernetes Master 节点保持业务方 stateful set/replication set 基础信息，平时其实例数为 0，大促前扩容实例，大促结束后再缩容至 0，如图 3-42 所示。

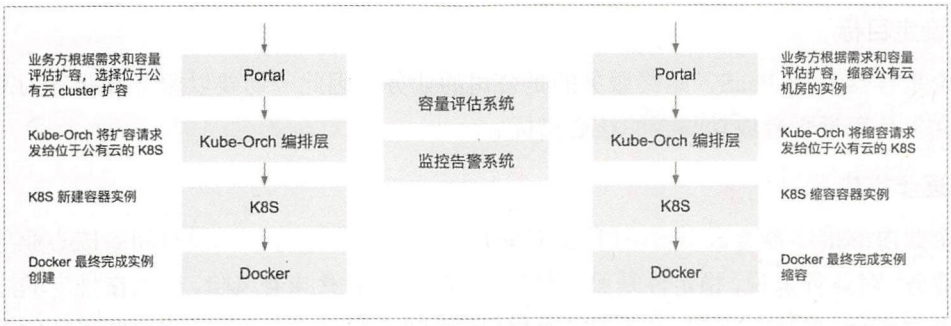


图 3-42　业务方扩容

PaaS 平台支持业务方指定机房扩缩容所需的实例如图 3-43 所示。

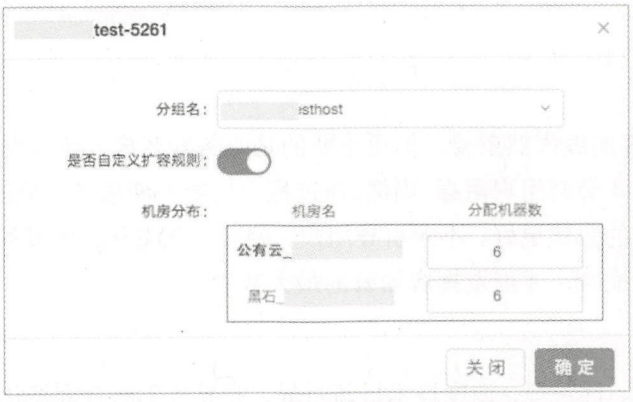


图 3-43　PaaS 平台支持业务方指定机房扩缩容所需的实例

从长期角度看，还将打通服务发现、SLB、监控告警模块，最终实现自动的弹性伸缩。

3.3　总结

回顾三年多来从无到有建设容器云平台的实践历程，期间遇到过很多问题，得到了一些经验和教训。这里总结了一些体会和心得，希望对大家能有所启发。

3.3.1　体会和心得

如图 3-44 所示为体会和心得结构图。

图 3-44　体会和心得结构图

1. 设定目标

无论是容器还是 PaaS，始终服务的是公司的业务，因此一切要以服务好业务为工作的出发点，这也是衡量平台成功与否的核心指标。

2. 逐步推广

在企业内部推广容器云平台，往往面对的是已经在线上运行了一段时间的核心业务或者非核心的业务。对业务来说，稳定性是第一位的，而改造业务意味着风险，所以在推广的过程中，不能追求大而全，要逐步推广。要采取“农村包围城市”的策略，寻找对容器感兴趣的业务方，找到合适的切入点，最好是他们想解决的痛点问题。然后以业务驱动的方式设定工作目标，层层推进。只有在帮助业务方真正解决了问题后，平台才会有生命力，后续的推广和平台建设也会水到渠成。

3. 如何切入

参考业界的 PaaS 实践固然重要，但更重要的是服务好客户。结合现实条件，设计和实现满足客户需求的平台才会有用户留存。当然，在过程中也要不断思考，哪些是业界推荐的做法，为什么这么做比已有的方式更好。不断引导用户，朝着无状态化、微服务化的方向演进。有时也需要业务方的改造配合，才能发挥容器云的最大威力。

4. 技术氛围

开源社区的容器、群管理、调度技术日新月异，业界也不断有容器化的落地案例诞生，比如 Pouch、KataContainer、gVisor 等。在容器云平台建设的过程中，不断学习业界的最佳实践，提升团队小伙伴的知识、技术能力，营造浓厚的技术氛围，才能让整个团队的成员有所成长。

如果能将新的知识和技术运用到实际的业务场景中，真正实践起来，才能让大家有所收获，同时真正为业务创造价值。

3.3.2 展望未来

容器技术还处于高速演进的阶段，我们重点关注以下几方面的技术演进，以及具有代表性的开源项目。

1. 容器隔离性

在实际的生产环境上，容器的隔离性是遇到最多的问题，比如磁盘异步 I/O 默认不支持隔离，最大进程数 pid_max 不支持隔离等，很多其实是内核层面由于历史原因，没有考虑过应用的容器化导致的。可以预见更强的容器隔离性，更好的稳定性是社区努力的方向，也确实诞生了一些有代表的开源项目。

1）KataContainer

KataContainer 是 Intel Clear Containers 和 Hyper runV 合并后的开源项目，完全兼容 OCI 标准，支持与 Docker 或 Kubernetes 无缝集成。2018 年 5 月 22 日，KataContainer 发布了 1.0 版本。如图 3-45 所示为 KataContainer 与现有 Docker 架构的对比。

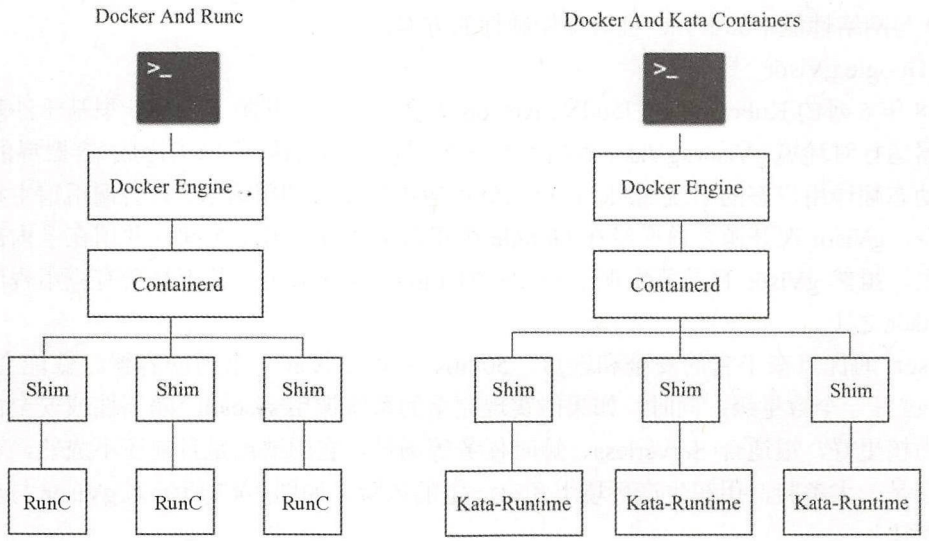


图 3-45　KataContainer 与现有 Docker 架构的对比

从技术角度来说，KataContainer 是基于全虚拟化的 QEMU 或 KVM 内核并兼容了 OCI 标准的镜像格式，通过快速启动一个轻量的虚拟机启动应用，如图 3-46 所示。

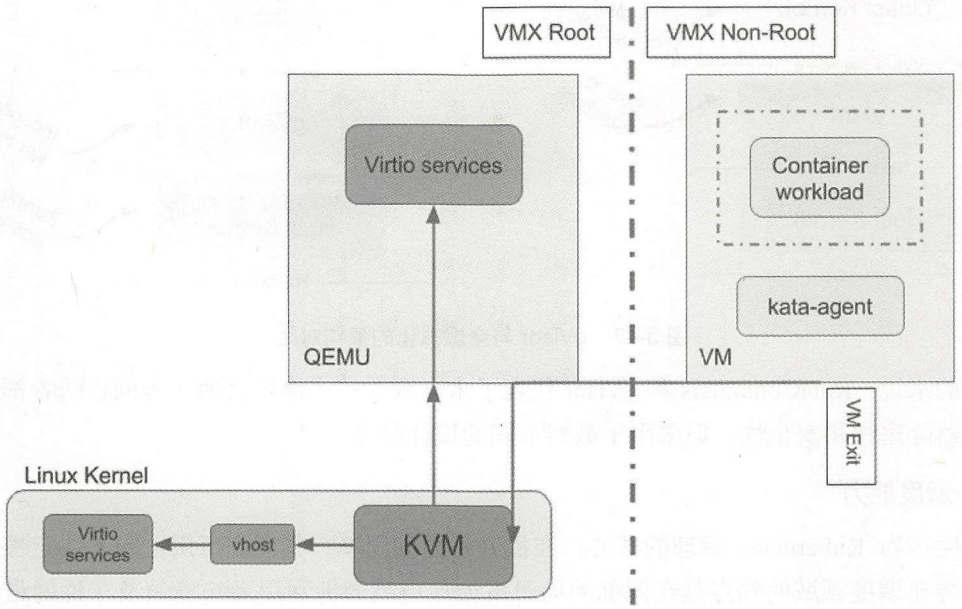


图 3-46　KataContainer

这么做的优点是很明显的。

- 强隔离性：因为完全是用与 KVM 一样的技术原理，不会由于内核隔离性的缺陷而产生问题。
- 安全性：KVM 靠硬件 CPU 的 VT-x 指令集实现全虚拟化，安全性比纯软件的实现更高，因此更加适合公有云的场景。

Kata-Containers 的缺点在于性能，KVM 全虚拟化的设计架构决定了在同等硬件条件下，

磁盘 I/O 与网络性能不如 RunC 直接操作硬件的方案。

2）Google gVisor

2018 年 5 月的 KubeCon + CloudNativeCon 大会上，谷歌开源了一种新型基于沙箱原理的开源容器运行时环境 gVisor。gVisor 本质上是一个用户态的内核，可以类比成“容器界的 JVM”。它通过动态翻译用户态的系统调用，自己来处理应用的系统调用请求，只会调用宿主机比较简单的指令。gVisor 在开源之前已经在 Google 公司内部研发了五年时间，并用在了内部的生产环境之上。虽然 gVisor 目前只实现了一部分的 Linux 系统调用，并不是所有应用程序都可以跑在 gVisor 之上。

gVisor 的优点在于它的安全和轻量，500ms 就可以拉起一个新的容器。性能会比 Kata Containers 好一个数量级。同时，如果能实现完全的系统调用 syscall，隔离性或安全性也会比现有的内核更好，很适合 Serverless、短时任务等场景。它的缺点是目前还不成熟，对应用的兼容性会是一大考验，用在生产环境上还有一定的风险。如图 3-47 所示为 gVisor 与全虚拟化的架构对比。

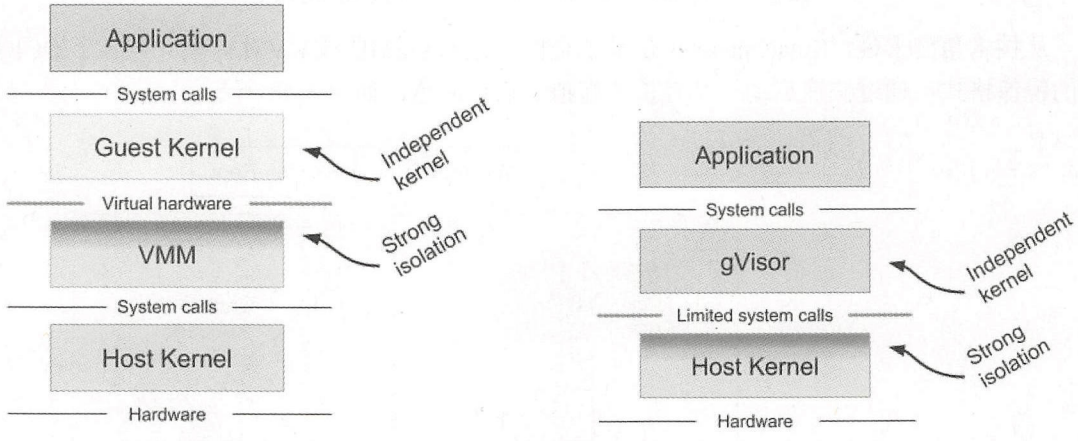


图 3-47　gVisor 与全虚拟化的架构对比

总的来说，KataContainers 和 gVisor 代表了未来容器技术演进的两个方向，两者都追求更强的容器隔离性和安全性，但采用了截然不同的设计思路。

2. 调度能力

调度作为 Kubernetes 管理的核心，在提升资源利用率、集群高可用上起了决定性作用。目前业界在调度领域的热点是在混部的场景下如何更高效地利用系统资源及在性能最优的场景下运行应用。Firmament 凭借优秀的调度算法能力开始进入我们的视野，Firmament 主要使用算法 Cost scaling 及 Relaxation 依据系统实时的性能数据进行决策，将任务调度至最优的节点，达到最大资源利用率，最优的运行性能的效果；在混部场景下对在线任务、离线任务及批量任务可以依据指定的调度策略来达到分时复用、抢占、亲和反亲和等特性，来高效地使任务在最小的资源利用的情况下性能最优。目前 Kubernetes 也积极与 Firmament 进行结合，开源项目为 kubernetes-sigs/Poseidon，美联也一直关注该项目的进展，并希望引进生成，丰富并优化 PaaS 平台的容器调度策略及性能。如图 3-48 所示为 Firmament 的架构示意图。

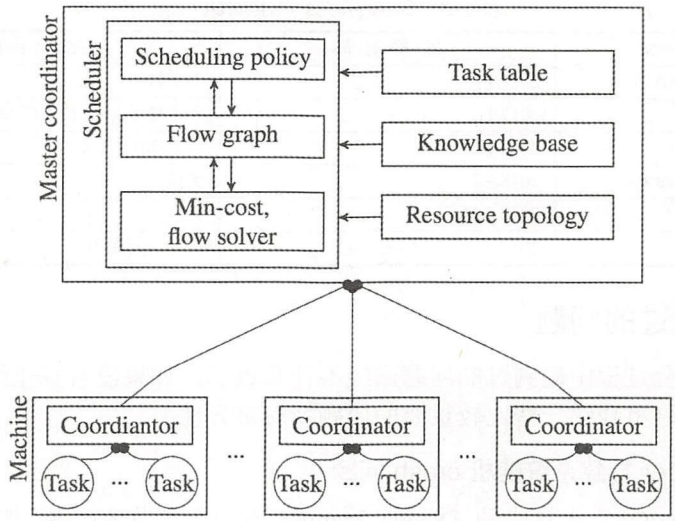


图 3-48 Firmament 架构示意图

3. 镜像构建

Docker 镜像一般是用 Dockerfile 进行构建和管理的，但这种镜像构建方式在业界也有不同的声音，认为它有一系列的问题，比如：

1）同一份 Dockerfile 在不同时间点打包出来的镜像可能是不同的，比如 yum install 的软件版本不同，二进制数文件是不同的。

2）镜像之间的依赖性问题。基础镜像如果更新了，所有依赖的镜像都需要重新构建，如果要替换线上的实例，需要重新发布所有的实例。

3）镜像里的操作系统和应用是静态绑定的关系。容易让用户把 Container 镜像变成“虚拟机”。

4）镜像内容的可视化不足。一旦打完了镜像，很难知道里面到底打包了什么内容。

因此业界诞生了不少开源项目，比如 Redhat buildah、Distroless、Buildkit、Source-to-Image、Habitat 等，希望解决上述的问题。

4. Serverless

无服务器运算（Serverless Computing）又被称为功能即服务（Function-as-a-Service，FaaS），是云计算的一种模型。Serverless 最早的实现可以追溯到 AWS Lambda 服务，它由事件驱动，应用的生命周期完全是被云服务商所管理的一种云服务。Serverless 的目标是让开发者不用过多考虑底层的计算资源，用户在服务部署级别来管理自身的应用程序。

容器的诞生，让用户和云服务商之间的接口变得更清晰和友好。用户只需要交付一个在本地测试通过、可运行的镜像或者 Dockerfile，剩下的事情由云服务商来做就行了。

随着类似 KataContainers 安全容器技术的诞生和走向成熟，Serverless 成为各大公有云厂商近年来着力发展的重点。2017 年，微软 Azure 首先推出了 Serverless 容器服务 CCI，紧接着 AWS、华为云、阿里云都推出了各自类似的容器云服务产品。CNCF 基金会也专门成立了 Serverless Working Group 研究 Cloud Native 与 Serverless 技术的结合点。如表 3-3 所示为 Serverless 产品对比。

表 3-3 Serverless 产品对比

Serverless Container	发布时间	Kubernetes 集成
Azure Container Instance （ACI）	2017.7	不支持
AWS Fargate	2017.11	不支持（AWS EKS 支持 Kubernetes）
Huawei CCI	2018.2	计划 2018 年支持
AliCloud Serverless （Bazaar）	2018.5.2	支持
Hyper.sh/pi	2018.5.11	支持
Kubeless	2018.5.28 v1.0.0-alpha4	支持（开源项目）

3.3.3 遇到过的问题

在建设 PaaS 平台过程中遇到过的问题已经不计其数了，如果没有真正经历过的人，则很难发现这些“坑”。这里介绍一些比较重要的问题及应对方法。

1. Device Mapper 导致系统随机 crash 问题

CentOS6 和 CentOS7 上自带的 Device Mapper 有一个严重的问题，它的 thin-provision discard 特性会造成内核的随机 crash。

```
Call Trace:
[<ffffffffa01e4876>] remove_raw+0x5f6/0x810 [dm_persistent_data]
[<ffffffff81058d53>] ? __wake_up+0x53/0x70
[<ffffffff81266c40>] ? generic_make_request+0x240/0x5a0
[<ffffffffa01e4b40>] dm_btree_remove+0xb0/0x150 [dm_persistent_data]
[<ffffffffa01fb727>] dm_thin_remove_block+0x87/0xb0 [dm_thin_pool]
[<ffffffffa01f7702>] process_prepared_discard+0x22/0x60 [dm_thin_pool]
[<ffffffffa01f6a37>] process_prepared+0x87/0xa0 [dm_thin_pool]
[<ffffffffa01f93e0>] ? do_worker+0x0/0x260 [dm_thin_pool]
[<ffffffffa01f9432>] do_worker+0x52/0x260 [dm_thin_pool]
[<ffffffffa01f93e0>] ? do_worker+0x0/0x260 [dm_thin_pool]
[<ffffffff81094d10>] worker_thread+0x170/0x2a0
[<ffffffff8109b290>] ? autoremove_wake_function+0x0/0x40
```

Docker 容器使用的存储是稀疏文件，镜像池文件虽然看上去有 100GB，但是实际占用的空间可能会很小，取决于容器内的文件占用空间。例如，用 ls-l 查看容器的镜像 data 文件大小为 100GB，但是用 du-hs 看空间占用可能只有 2GB，这就是稀疏文件，未使用的空间都用 0 来填充。discard 的功能是当容器内删除文件时，文件池会释放这部分空间，减小空间占用。在删除整个容器的镜像时，discard 功能也会发挥释放占用空间的作用。discard 功能是在内核 2.6.X 版本中引入的，在 3.8 中 dm-thin 支持了 discard。美联使用的 CentOS 也有 dm-thin 的 discard 功能。解决（规避）整个问题的思路是，通过关闭 discard 功能，从而就不会走到 kernel crash 的那段代码流程。

规避该问题的办法是关闭 discard 功能，在 Docker daemon 启动参数中添加如下代码：

```
DOCKER_STORAGE_OPTIONS="--storage-opt dm.mountopt=nodiscard --storage-opt dm.blkdiscard=false"
```

参考链接：https://tech.meili-inc.com/docker_crash-79。

2. Kernel forkbombs

在 Linux 内核中，/proc/sys/kernel/pid_max 一直是一个全局的进程数量上限。有一次业务方在容器内的脚本有 Bug，会不停地创建进程，最终会导致容器和宿主机无法登录，因为 SSH Client 登录也会在宿主机上新创建进程，当全局的 PID 资源耗尽时，对整台机器的影响是致命的。

规避该问题的方法有两个：

1）将内核版本升级至 4.3，这里是 Linux Kernel 4.3 cgroup pids subsystem 的 patch：https://patchwork.kernel.org/patch/6571481/。Linux 内核关于 process number controller 的说明：https://www.kernel.org/doc/Documentation/cgroup-v1/pids.txt。

2）增加针对活跃进程数的监控，当超过一定的阈值，比如 80%时，进行报警。

3.3.4 开源工具分享

“工欲善其事，必先利其器。”解决底层复杂的系统性问题，必须要能找到更强大、更高效的工具快速定位问题和解决问题。很多时候，也是问题逼迫我们必须去寻找，甚至开发出工具应对问题。这里分享几个开源工具。

1. 系统问题分析

1）perf-tools

perf-tools 是 Brendan Gregg 开源的 Linux 系统性能分析和跟踪的工具集合，如图 3-49 所示。这套工具集里有非常多值得借鉴和利用的工具，比如：

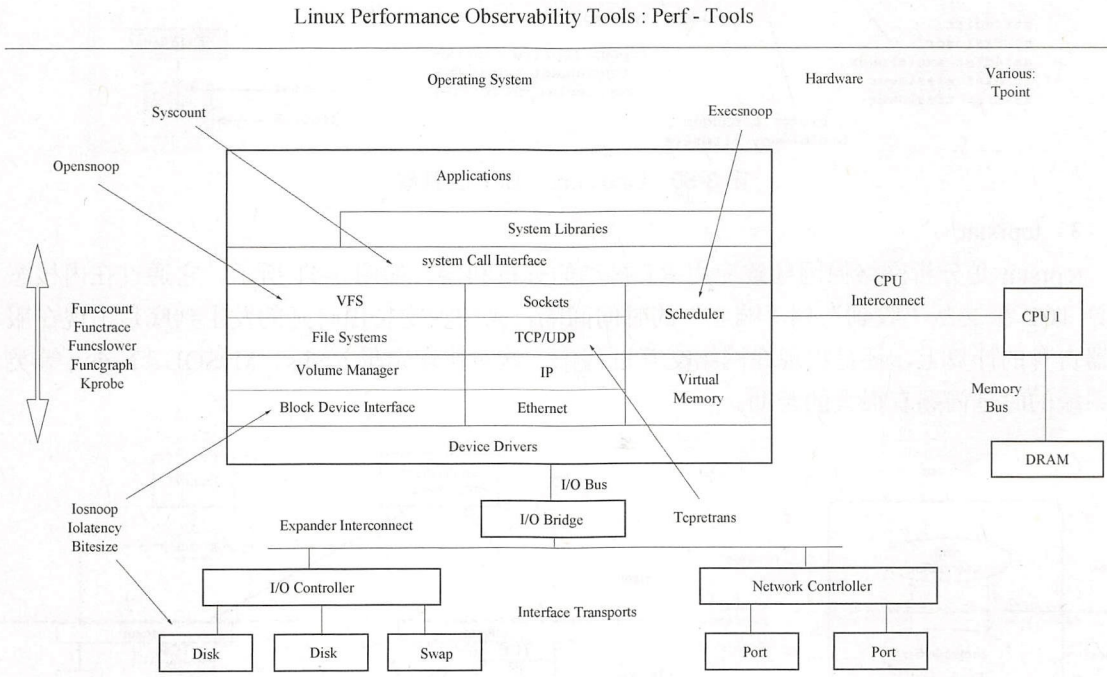


图 3-49 perf-tools

- tcpretrans：实时跟踪并展示发生 TCP 重传的 TCP 连接。当本机的重传率上升时，它能很方便地展示出对方服务器的 IP 地址，从而快速定位问题。
- iosnoop：跟踪进程（含应用名和 PID）的磁盘 I/O 请求量及 I/O 响应时间。
- iolatency：图形化展示磁盘 I/O 时间消耗分布。
- killsnoop：跟踪 Linux 内核进程间发送的信号，在进程异常退出时，可以找到哪个进程发出了什么信号导致它退出。

2）Linux bcc / BPF 工具集

eBPF（extended Berkeley Packet Filters）是 Linux v3.15 之后推出的新的内核跟踪框架，如图 3-50 所示。BCC 是基于 eBPF 的一套工具集，能让用户通过编写 C 和 Python 代码，就能很

容易完成对内核行为的跟踪。

从图 3-50 可以发现，这套工具集合能够定位很多系统问题。

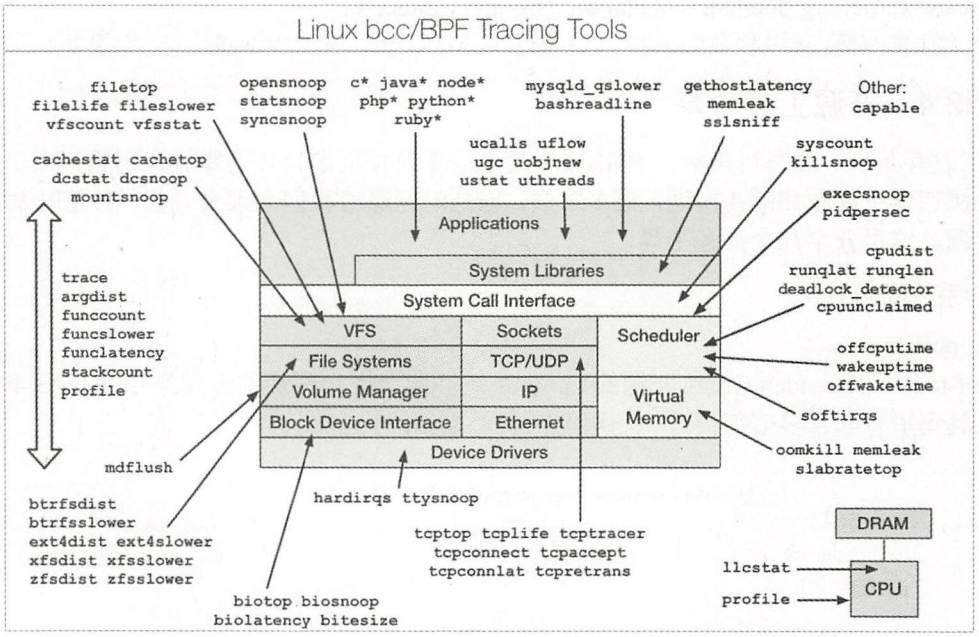


图 3-50　Linux bcc / BPF 工具集

3）tcprstat

tcprstat 是分析网络原因导致应用 RT 抖动的分析利器，如图 3-51 所示。它通过在内核态计算 TCP 报文从“收到”到“响应”的时间间隔，来快速定位出时延的发生到底是出现在服务器自身的计算上，还是出现在网络交互过程中。这对快速定位 Redis、MySQL、Nginx 等类似系统的时延问题有很大的帮助。

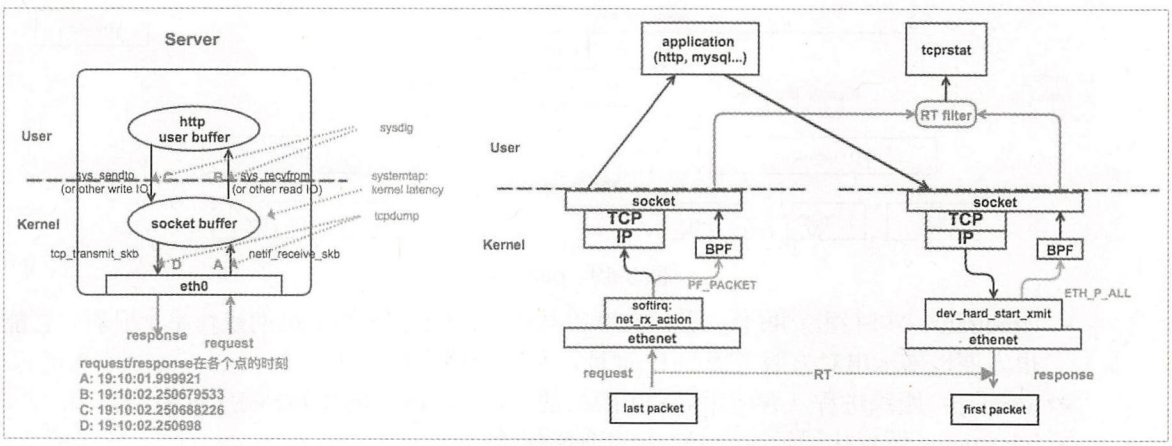


图 3-51　tcprstat

在服务器端，具体使用命令为：

```
$ tcprstat -l <server_ip> -p 6379 -T 1
```

其中，-l 后面是服务器自己对外通信的 IP，-p 6379 表示监听 TCP 服务器端口号为 6379 的通信，-T 1 表示命令行打印出超过 1ms 延时的通信。

在客户端，具体使用命令为：

```
$ tcprstat -d <server_ip> -p 6379 -T 1 -l <client_ip>
```

其中-l 后面是客户端对外通信的 IP，-d 后面是与之通信的服务器 IP，-p 6379 表示监听 TCP 服务器端口号为 6379 的通信，-T 1 表示命令行打印出超过 1ms 延时的通信。

在定位 TCP 延时的网络问题时，通常在服务器和客户端两侧都启动该命令进行分析。

如果客户端网络 RT 非常高，而服务器网络 RT 没有问题，那么问题就可能出在外部网络，如交换机、路由器；如果客户端和服务器网络 RT 都非常高，那么就是服务器响应网络请求过慢，就要从服务器自身找问题。

前文介绍了 Docker 外面的组网有用到 OVS，而且用到了 Vlan 模式，实际上 Docker 所有的包都要通过 OVS 向外通信，那么在宿主机上用 tcprstat 命令也可以直接分析 Docker 的网络延时问题。所以，我们对 tcprstat 做了改造，使其支持了 Vlan 模式的报文分析，而且由于增加了一个网络分析的点，所以更能清楚地找到是网络的哪个段出了问题。未改造之前的网络分为三段：客户端网络、服务器网络、外部网络；改造后的网络分为五段：客户端 Docker 网络、客户端 Docker 的宿主机网络、外部网络、服务器端 Docker 网络和服务器端 docker 的宿主机网络。

本章作者：张振华，花名郭嘉。

参考文献

[1] 美丽联合技术博客：https://tech.meili-inc.com/.

[2] Kubernetes Pod Eviction 驱逐策略：https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/https://www.ibm.com/support/knowledgecenter/SSBS6K_2.1.0/installing/pod_eviction.html.

[3] OpenStack Kuryr：https://github.com/openstack/kuryr.

[4] 记一次k8s集群单点故障引发的血案：https://blog.csdn.net/yuedaoshengsi/article/details/54883039.

[5] Kata-Containers：https://katacontainers.io/.

[6] gvisor：https://github.com/google/gvisor.

[7] CNCF Serverless Working Group：https://github.com/cncf/wg-serverless.

[8] Survey of Container Build Tools：https://schd.ws/hosted_files/kccncna17/d8/CNCF%252-FKubecon%20NA%202017%20-%20Survey%20of%20Container%20Build%20Tools%20%282%29.pdf.

[9] perf-tools: https://github.com/brendangregg/perf-tools.

[10] Linux bcc / BPF 工具集，https://github.com/iovisor/bcc.

[11] tcprstat: https://github.com/Lowercases/tcprstat.

[12] firmament: https://github.com/camsas/firmament.

[13] Poseidon: https://github.com/kubernetes-sigs/poseidon.

[14] K8S 本地存储容量隔离，https://github.com/kubernetes/kubernetes/issues/43607.
[15] https://github.com/kubernetes/community/pull/306.
[16] http://blog.csdn.net/waltonwang/article/details/55804309.
[17] https://www.ibm.com/support/knowledgecenter/SSBS6K_2.1.0/installing/pod_eviction.html
https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/#kubelet-may-not-observe-memory-pressure-right-away.

CHAPTER

4

第4章 酷家乐容器化之路

酷家乐是一个被广大业主、设计师、商家喜爱的在线家居设计平台。独特的业务场景决定了酷家乐不同于电商网站或社交网络的流量模型：春季高峰、高并发、大部分设计工具内的API有I/O密集或CPU密集的特性。

一方面是特殊的流量模型，一方面是酷家乐用户数据每年迅猛的增长，在酷家乐工程团队几年来的实践中，解决了很多其他团队碰到过的问题，也遇见过没有现成经验、需要自己摸索解决的问题。其中，“容器化改造”是一个我们付出了大量努力、效果比较好，且容易向外界技术团队提供经验的技术话题。

酷家乐内部将容器化改造划分为三个阶段，简称“三步走”。“三步走”的具体内容会在第一部分内容中讲解，分别是“应用容器化”“编排自动化”“服务网格与混合部署”。

酷家乐目前完成了第一步和第二步，第三步正在积极探索当中。本文的作者都是酷家乐团队容器化改造一线的工程师，以不同的角度深度参与了酷家乐容器化改造的进程。我们很荣幸可以将积累到现在的经验知识分享出来，给后续对容器化改造有需求和感兴趣的技术团队参考。

本章的主要结构按照上述三步走的顺序展开，介绍其中的技术点与经验教训，重点是第一步和第二步，最后一部分是酷家乐在服务网格上的实践，可以算作第三步的一部分内容。

4.1 架构挑战与应对方案

2016 年年初，由于业务流量与复杂性的双重爆发性增长，最早的单体架构已经难以为继，酷家乐开始了全站的服务化改造。此后一年半内，随着服务化改造的逐步深化，服务粒度不断变小、数目不断膨胀。截至 2017 年年中，启动容器化改造之前，线上已经有百余个不同的应用，这对整个开发部署运维流程的各个环节都带来了前所未有的挑战。在这个关头，酷家乐选择使用容器化技术解决种种问题，并开启了全业务线的容器化改造。

酷家乐的线上服务采用了服务化架构（Service-oriented Architecture，SOA），总体包含百余个应用。每个应用通过暴露 RESTful API 供外界调用，并在服务注册中心上报自己的 IP 以及提供的 API。API 调用方通过注册中心订阅需要调用的服务来实现服务发现。整个系统通过 API Gateway 将部分 API 暴露给外部，同时，Gateway 上还会进行 Session 管理、认证、限流等操作。所有 SOA 体系内的线上服务都部署在云服务商的虚拟主机上。整体 SOA 架构如图 4-1 所示。

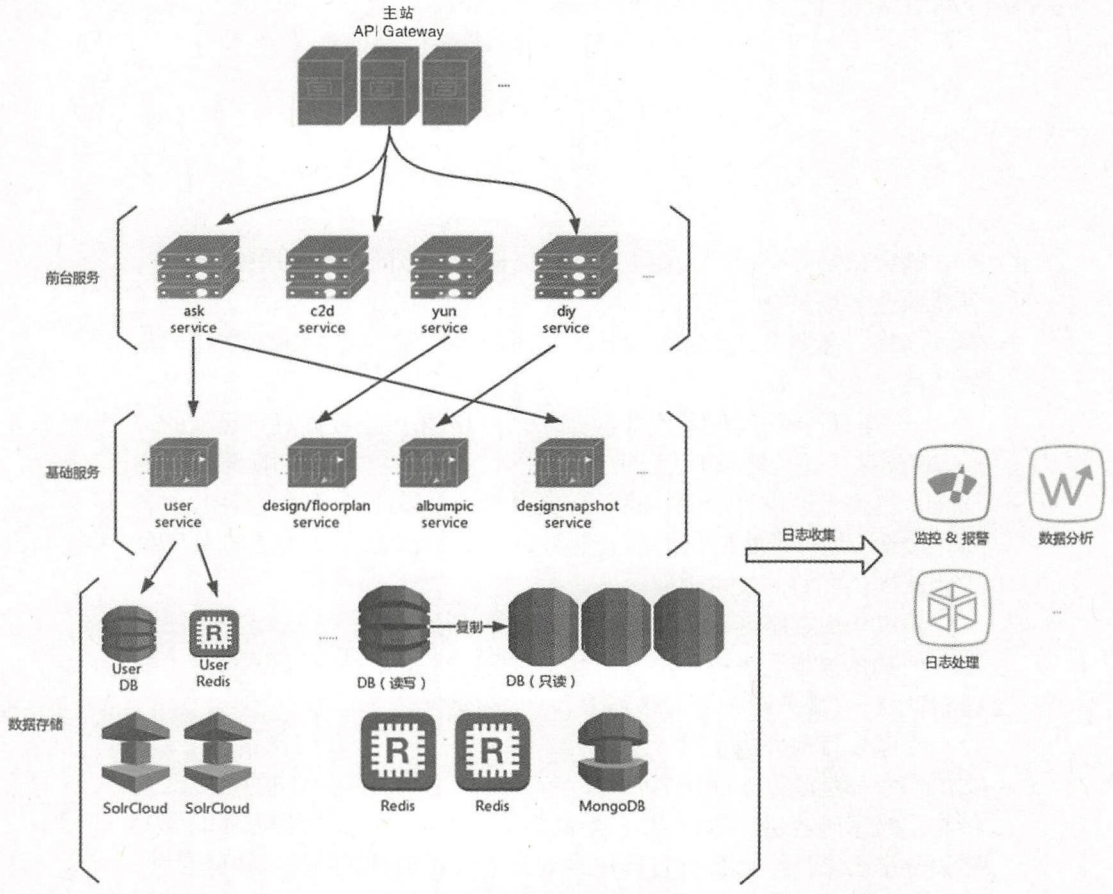


图 4-1 整体 SOA 架构

在业务迭代的不断驱动下，应用的粒度还在不断变小、数目不断增加，相应需要的虚拟主机数目也快速增加。这带来了几个问题：

1. 资源利用率低

每个应用实例都需要申请一台单独的虚拟主机部署。同时，为了满足灰度发布和高可用，

一个应用无论流量多小，都至少需要 N*2 个实例（N 为灰度发布时同时存在的版本数量）。因此，在大应用不断被切分成多个小粒度应用的情况下，不得不购买大量的低配虚拟主机，虚拟主机的整体资源利用率不断降低。

2. 管理困难

截至 2018 年 5 月，酷家乐已经维护着近千台虚拟主机。管理大量虚拟主机与大量应用间的映射关系已经给研发与运维带来了很大的负担。同时，在 DevOps 模式下，直接面向虚拟机也让运维人员和研发人员的职责切分不够清晰。

3. 运维困难

每个应用都对应着一组固定的虚拟主机，在扩容和缩容时，虚拟主机的购买、释放、初始化都需要人工介入，周期漫长。在此基础上，还需要为每台虚拟主机维护日志收集与监控报警的 Agent，以及某些应用特有的依赖程序，这让整个问题变得更为烦琐。尽管写了大量的自动化脚本进行辅助，仍免不了在其中某环上发生错误，严重的还可能导致线上故障。

伴随着 Kubernetes 的快速成熟，酷家乐选择将线上应用逐步迁移到 Kubernetes 解决这些问题。Kubernetes 解决了三个层面上的问题。

1）容器化：将应用的交付模式从传统的二进制包转换到标准镜像，如 Docker Image。这使得应用的运行环境和应用本身绑定，大大简化了应用的交付与运维。

2）调度：解决了如何为容器分配运行时资源的问题。轻量级的虚拟化技术外加统一资源调度，大大提升了部署密度，优化了资源利用率。同时，对于 DevOps 的支持更加彻底。

3）服务抽象：Kubernetes 通过 Deployment、Service 等概念构建了一个完善的微服务体系。同时，用声明式 API 将部署、扩容、缩容等操作标准化，让 CI/CD 与监控报警等周边系统的整合变得非常简单。

但是，在解决上述问题的同时，Kubernetes 本身也引入了巨大的复杂性——它本身是一个非常复杂的分布式系统，提供的 Service 抽象又与 SOA 体系不能完全兼容，贸然迁移到 Kubernetes 会有很大的技术风险。同时，产品的迭代是不可能因为技术改造而停滞的，因此整个容器化进程不能影响产品的快速迭代也是容器化改造的大前提。

为此，需要设计一个平滑稳定的迁移方案，逐步进化到基于 Kubernetes 的容器化架构，最小化其中的风险与对业务迭代的影响。经过一系列的内部讨论，最终敲定了整个容器化进程的“三步走”策略。

第一步，容器化：直接在裸机上运行 Docker 容器，一个虚拟主机对应一个容器。该阶段的目标是将应用的交付模式从程序包转变为 Docker 镜像。这个过程对于现有的开发部署流程，以及 CI/CD、监控报警等系统的改造都很小，因此风险可控。同时能够让公司整体熟悉 Docker 相关的部署运维工具。

第二步，编排自动化：逐步将应用迁移到 Kubernetes 上，而微服务架构本身仍然沿用现有的 SOA 方案。这一阶段的目标是用 Kubernetes 解决资源利用率和 DevOps 两大问题。在第一步的基础上，这一步只是把运行时从虚拟主机交给了 Kubernetes，风险仅仅在于可能碰到 Kubernetes 本身的问题或漏洞。而这一风险可以通过一段时间的裸机运行 Docker 与 Kubernetes 并存来进行规避。

第三步，服务网格与混部：这一阶段，将基于 Kubernetes 提供的编排能力与 Service 抽象实现两个目标：一是将现有的 SOA 体系与 Kubernetes 进行整合，最终演化到服务网格架构；二是实现在线离线混部，进一步提升整体的资源利用率。

在后面的内容中，将分别阐述三步走战略中每一步遇到的问题以及解决方案。

4.2 应用容器化

本节将介绍在应用容器化进程中，应用本身镜像化以及对 CI/CD 系统的调整，网络和服务注册中心的调整情况。

4.2.1 CI/CD 迁移

在启动容器化计划之前，酷家乐已经有一个与 CI/CD 系统实现相关的流程。容器化计划是容器技术第一次在酷家乐业务中大规模使用，影响最紧密的基础设施就是 CI/CD 系统。酷家乐很多工程师也在这个阶段才开始真正接触容器技术，需要一段时间学习相关的核心技术。CI/CD 系统在整个容器化进程中，既是一个重要的承接者，也是一个强力的推动者。

当时酷家乐 CI 的主要内容是单元测试和静态检查，CD 的主要内容是构建、部署和 API 测试。承载这些 CI/CD 内容的基础设施都是传统的虚拟机，比如单元测试是运行在 Jenkins 的 Slave 机器上的，而应用都是运行在虚拟机里的。根据应用框架的不同，在虚拟机上安装配置不同的应用环境，比如 Jetty、Node.js、Spring Boot 等。CI/CD 系统也是以此为基础建设的，为了支持容器化，单元测试、应用构建、部署等底层设施需要新的解决方案；而静态检查和 API 测试由专门的系统负责执行，不需要应用的运行环境可以不做改变。

最终决定采用 Kubernetes 的 Job Controller 来做单元测试和构建任务，相比于原来使用 Jenkins 的方案，Kubernetes 主要有以下几个优势：

- 高可用，没有单点故障；
- 可扩展性好，增加节点非常简单，轻松做到水平扩展；
- 天然支持容器化；
- Job Controller 自带重试机制，简化容错的逻辑；
- 可对每个任务进行资源限制，防止单个不合理的任务耗尽机器资源。

在容器化第一阶段，酷家乐确定了每台虚拟机上只运行一个应用的容器实例。在这样的限制条件下，决定让部署系统直接连接主机上的 Docker Daemon 实现对单个实例的部署。

除了技术方案的确定，还确定了容器运维体系的方案和相应的规范。当考虑这个方案的时候，首先需要考虑的是运维和开发的职责界限。酷家乐的整体技术氛围偏向 DevOps 文化，没有专门的应用运维，应用的常规运维操作如部署、重启、扩容等都由业务工程师自己完成，而运维工程师负责提供基础设施和平台帮助业务工程师完成运维操作。基于这样的体系，规范大致包含以下内容：

- 统一运行应用的 Docker 版本和配置；
- 启动容器时的参数，包括网络模式、环境变量等；
- 要求每个应用用代码声明自己的运维属性，至少包括构建可发布镜像的 Dockerfile 和健康检查的方法。一些额外的属性或配置也用代码声明，并在 VCS 中管理起来。

4.2.2 公共基础镜像

容器化首先遇到的一个问题是如何选择基础镜像的操作系统，Alpine 镜像及其精简的大小是其巨大的优势。不过对于酷家乐来说，镜像大小不是关键，功能完整性和易用性才是需要关注的。并且，Docker 镜像的分层设计使得镜像大小在大部分时候都不是什么问题。因此，最终选择了有较多运维经验的 CentOS 作为基础镜像的操作系统。

为了满足不同的业务需求，建立了一个公共镜像库提供给全公司使用，整个镜像库有明确的层次结构，每个公共镜像提供特定的功能，并且都是由最基础的 CentOS 镜像衍生出来的，保证基础的配置和优化能迅速地在全公司内生效，整个结构如图 4-2 所示。

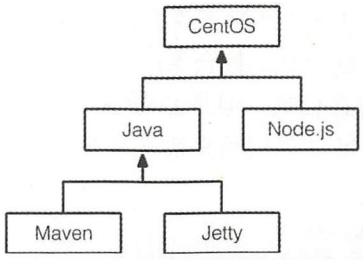


图 4-2　CentOS 结构

4.2.3　镜像构建及单元测试

在 CI/CD 中，构建镜像和单元测试是非常频繁的任务。这些任务要求的计算资源往往不少，并且希望运行速度越快越好。随着业务的发展，这些任务的量也会越来越大。CI/CD 系统原本依托于 Jenkins 运行构建任务，现在则是直接使用 Kubernetes 的 Job Controller 作为构建任务底层设施。

Job Controller 的基本原理是创建一个或多个 Pod，并确保一定数量的 Pod 成功地完成。在酷家乐的使用场景下，创建一个只包含一个 Pod 的 Job，确保这个 Pod 能成功完成。比如下面这个简单的例子，是一个用来构建镜像的 Job，用专门的镜像完成构建任务，并用环境变量传入参数。由于需要用到 Docker，将 Node 的 Docker socket 挂载到了容器里面，这样 Pod 就可以直接使用 Node 的 Docker daemon 了。这个 Pod 做的主要工作就是运行 docker build 生成镜像并上传。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: build-image
spec:
  template:
    spec:
      containers:
      - name: builder
        image: kjlcr.com/builder:1.1.3
        env:
        - name: REPOSITORY
          value: git@github.com:kujiale/foobar.git
        - name: BRANCH
          value: master
        volumeMounts:
        - mountPath: /var/run/docker.sock
          name: docker-socket
      volumes:
      - hostPath:
          path: /var/run/docker.sock
          type: Socket
        name: docker-socket
```

对于单元测试或者其他可能的任务，也用类似的 Job 执行。Kubernetes 自带的资源限制功能在这样的场景下也变得非常适用，为每个 Job 设置一定的 CPU 和内存限额，以防止有不合理的任务拖垮整个 Node。

解决了任务调度的问题后，镜像构建还有一个技术问题需要解决。构建工具往往会在本地

缓存一些文件来加速构建，比如 Maven 会把下载过的依赖缓存到本地仓库。这样的缓存在使用 Jenkins 时一切正常，运行 Maven 产生的缓存文件会直接存在 Slave 的机器上。但是当用 Docker build 直接运行 Dockerfile 构建时，就没法使用这样的缓存。比如说在 Job 中 Docker build 下面的 Dockerfile，每次 Docker build 都要下载大量的 Maven 依赖文件，总量可达到几百兆，使得整个构建过程变得非常慢，也会对依赖仓库产生巨大的压力。要解决这个问题，就得在运行 Docker build 时挂载共享的缓存目录，但是 Docker 官方并不支持并且不打算支持这样的功能。因此，我们修改了 Docker build 的源代码来实现在 build 时挂载主机上的某个目录的功能，保证镜像构建可以快速地运行。

```
FROM kjlcr.com/maven:3.5 as builder
RUN mvn clean package
FROM kjlcr.com/jetty:9.4
COPY --from=builder /workspace/target/app.war /var/lib/jetty/webapps/root.war
```

4.2.4 容器部署

由于 Docker 提供了一套标准的 API 操控容器，部署本身的实现是比较简单的。我们使用分布式任务队列 Celery 实现部署任务，在每个机房部署若干个 Worker 部署对应机房的服务器。

在实际迁移过程中，我们希望整个过程尽量平滑，对开发透明。为此，先在主机上安装好 Docker 环境，并且部署任务支持在一台主机上部署传统应用或者容器应用，构建任务也同时生成原来的软件包和容器镜像。这样，当在部署某个服务时，可以自由地切换是否部署成容器，以应对迁移过程中的各种问题。

4.2.5 网络模式

完成了应用交付与 CI/CD 的改造后，还需要保持对现有的日志收集、监控报警以及网络模型的完全兼容。Docker 容器的本质是加了 Namespace 隔离和 Cgroup 资源限制的进程组，因此基于进程信息收集资源使用量并上报的监控 Agent 不会有问题。但在 SOA 体系中，应用需要向注册中心上报自己的 IP，在默认的网络模式（Bridge 模式）下，Docker 会为容器分配一个独立的 Network Namespace，导致应用中获取 IP 可能不正确。为此，我们通过 host 模式启动容器：

```
docker run --network=host
```

与此同时，在 Bridge 模式下通过 DNAT 进行端口映射会有一定的网络损耗，而 host 模式则没有这个问题。可以说该模式完美契合了在第一阶段下的需求：仅仅修改应用的打包和部署方式，其他环节尽量保持不变。

4.2.6 性能相关

另外，Docker 化的性能损耗也是在这一过程中密切关注的问题。尽管业界已经有性能测试报告，显示 Docker 容器在各方面的性能损耗都是非常低的，但由于实际生产环境的情况往往更加复杂，我们进行了一系列的性能测试，对比了关键服务在容器化前后的性能指标。最终结论如下：

- 低负载下，容器化前后接口性能表现无明显差异；
- 高负载下，容器化前性能略优于容器化环境；

4.2.7 小结

第一阶段以一个较小的步伐展开容器化的推进，以使整个技术改造能有一个平滑的过渡期。在这个阶段，整个团队获得了容器相关的技术积累和基础技能，包括一个稳定基础镜像库、Dockerfile 编写方法、镜像和容器相关基本原理、容器中 Debug 的技能等。这些积累为接下来的两个阶段打下了坚实的基础。

4.3 编排自动化

完成应用容器化后，只解决了三大痛点中的半个——运维困难。其中，虚拟主机的配置维护问题顺利被 Docker 解决了，但仍存在资源利用率低、管理困难以及运维中的扩容缩容麻烦、日志 Agent 难以维护等问题。为了继续解决这些问题，正如之前的“三步走”策略所规划的，酷家乐开启了 Kubernetes 迁移工程。

在开始迁移前，我们面临的第一个问题就是：选择自建 Kubernetes 集群还是购买云服务商的 Kubernetes 服务？总体来看，自建 Kubernetes 集群在初期需要投入相当的研发成本，并且一开始的稳定性也难以与云服务商提供的 Kubernetes 比肩。但长期来看，云服务商提供的 Kubernetes 难以定制，并且依赖于云服务商的服务，酷家乐自身将难以进行 Kubernetes 的技术积累，可能产生 Vendor Lock-in（厂商锁定，即对特定云服务厂商产生技术依赖）。另外，除了所购买的公有云服务外，酷家乐自身也维护着多个 IDC 机房，其中不乏渲染、监控、大数据、机器学习这样的核心业务。自建 Kubernetes 所积累的技术经验能够快速惠及所有 IDC 机房的运维调度。这些业务的特征是计算量极大且峰谷明显——比如渲染业务在下午会迎来流量波峰值，而在午夜则通常是波谷期，大数据或机器学习则完全可以利用这部分处于波谷期的大量 GPU 资源。因此接入 Kubernetes 不仅能在短期优化部署运维，还能为接下来的资源混部节省大量的资源。对于酷家乐而言，在 Kubernetes 上做技术投资杠杆率极高。

基于上面的考虑，最终决定在 IDC 和云服务商的虚拟主机上都自建 Kubernetes，并加大技术投入进行技术积累。未来，假如云服务商提供的 Kubernetes 服务整体的性价比足够诱人，也不排斥自建 Kubernetes 与云服务商的 Kubernetes 服务共存——此时我们已经有了足够的技术积累，部分采用云服务 Kubernetes 是从成本出发的结构优化，而不会再对云服务商产生技术依赖。

本节将介绍酷家乐如何将 Kubernetes 落地到实际的业务中，重点包含 Kubernetes 的资源隔离、权限控制、Kubernetes 的包管理、网络方案、存储方案、CMDB 改造、日志与监控等。

4.3.1 资源隔离与资源限额

搭建 Kubernetes 集群后，我们首先在内部系统以及国际化、实时流计算等较新的线上系统进行了初步尝试。很快就碰到了第一个问题——开始大家都没有使用经验，提交给 Kubernetes 的 workload 中都没有声明请求的资源与资源的最大限额。这会导致两个问题，一是 Kubernetes 无法预估 Pod 的资源用量，可能将 Pod 调度到一个资源吃紧的节点上；二是 Pod 的资源用量无限制，出现问题可能会拖垮节点。与此同时，第二个问题也很快浮出了水面，当时在 IDC 机房的 Kubernetes 集群上，CI 任务、流计算任务、微服务、监控报警组件全都混部在一个机器池下，Pod 密度非常高。各个业务方在服务变慢甚至节点被拖垮时都认为是其他业务影响了自身。

此时，可以选择自研或增强容器引擎，强化隔离性来从根本上消除上面的问题。这种方案

虽然从技术上来看非常优雅，但是对于当时的酷家乐而言，投入产出比是很低的。事实上，可以从调度的层面出发，借助一些 Kubernetes 本身的特性，就能以很低的代价解决上面两个问题，把注意力重新放回编排自动化这件事本身上。

解决方案就是准入控制器（Admission Controller）。准入控制器是 Kubernetes 的 APIServer 上的一个链式 Filter，它根据一定的规则决定是否允许当前请求生效，并且有可能会改写资源声明。假如整个处理链中有任何一个准入控制器拒绝了当前请求，那么请求被抛弃；反之，通过了所有准入控制器的请求则能被 APIServer 处理。为了解决上面的问题，需要用两个准入控制器：

PodNodeSelector 控制器的作用是限制每个 Namespace 中的 Workload 能够使用的 NodeSelector，并设置默认的 NodeSelector。NodeSelector 是 Kubernetes 的一个重要概念，它用于声明一个 Pod 可以调度到哪些节点上，最简单的一个例子是：

```
spec:
   containers:
   - name: nginx
     image: nginx
   nodeSelector:
     disktype: ssd
```

该 Pod 将只会调度到符合 label disktype=ssd 的节点上。我们可以为每个 Namespace 限定固定的机器池，打上不同的 Label，然后使用 PodNodeSelector 为每个 Namespace 下的 Pod 自动添加 NodeSelector。这样一来，每个 Namespace 下的 Pod 只会选择固定一个机器池，实现了 Namespace 间的高度资源隔离。PodNodeSelector 的配置方法如下：

首先，我们需要在 APIServer 的启动参数中，添加 PodNodeSelector。

```
/usr/bin/kube-apiserver
   --admissioncontrol=PodNodeSelector…
```

接下来，可以通过中心化配置的方式或 Namespace 注解的方式为各个 Namespace 声明默认的 NodeSelector。

全局配置：

在 APIServer 的启动参数中增加一个参数—admission-control-config-file=admission-config.yaml 用于指定 Admission Controller 的配置。再在该 YAML 中配置 PodNodeSelector 配置文件的位置：

```
kind: AdmissionConfiguration.
apiVersion: apiserver.Kubernetes.io/v1alpha1
plugins:
- name: PodNodeSelector
  path: pod-node-selector.yaml
```

接下来在对应的 pod-node-selector.yaml 中声明规则。

```
podNodeSelectorPluginConfig:
 clusterDefaultNodeSelector: <labels>
 namespace1: <node-selectors-labels>
 namespace2: <node-selectors-labels>
```

一目了然，第一行定义了全局的默认 NodeSelector，而后面几行定义了每个 Namespace 中的 Workload 默认的 NodeSelector。

注解配置：

除上面的方法，PodNodeSelector 还支持在 Namespace 的注解中声明该 Namespace 对应的 NodeSelector。

```
apiVersion: v1
kind: namespace
kind: Job
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/node-selector: "env=test,infra=fedora"
  name: namespace3
```

上面的 Namespace 资源文件声明了该 Namespace 的默认 NodeSelector。基于我们的实践，推荐第二种注解配置的方式。因为在开发人员提交一个 Merge Request 创建 Namespace 时，这种方式更容易进行代码审查。

那么，有了 PodNodeSelector 之后，是不是意味着需要给每个 Namespace 都配置一个机器池了呢？当然不是这样的。机器池隔离固然能够带来很强的隔离性，但也会显著降低整个集群的混部密度，降低整体的资源利用率。试想，假如隔离到最后变成一个 Pod 对应一个节点，那编排还有什么意义呢？因此，酷家乐的实践是适度使用机器池隔离，即针对特殊的服务——如可能占用大量资源的 CI 任务单独或是优先级特别高的核心业务——单独提供一个机器池进行隔离。其他 Namespace 则全部使用同一个公共机器池。哪些业务需要隔离，哪些业务可以混部，都需要依据实际的业务场景进行分析，涉及资源利用率和隔离性的取舍，不可一概而论。

第二个要介绍的准入控制器是 LimitRange。它的作用是为 Pod 添加默认的 Memory/CPU Request 和 Memory/CPU Limit，在 APIServer 上启用该控制器后，就使用 LimitRange 资源为 Namespace 指定默认的资源限额，以及每个 Pod 最大和最小所能申请的资源。比如，可以创建一个下面的 LimitRange。

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
  namespace: namespace3
spec:
  limits:
  - default:
      memory: 500Mi              // 默认的 Memory Limit 为 500MB
      cpu: 1                     // 默认的 CPU Limit 为 1
    defaultRequest:
      memory: 500Mi              // 默认的 Memory Request 为 500MB
      cpu: 1                     // 默认的 CPU Request 为 1
    max:
      memory: 2Gi                // 假如 Workload 自己指定 Memory Limit，那么不能超过 2GB
      cpu: 2                     // 假如 Workload 自己指定 CPU Limit，那么不能超过 2
    min:
      memory: 100Mi              // 假如 Workload 自己指定 Memory Request，那么不能低于 100
      cpu: 0.5                   // 假如 Workload 自己指定 CPU Request，那么不能低于 0.5
    type: Container
```

上面的 LimitRange 会使所在的 Namespace 获得限制：

- 默认的 CPU 和内存的申请额度是 1 核 500MB，限额也是 1 核 500MB。
- 假如 Workload 指定 Resources Limit 与 Request，那么最大的资源限额不能超过 2 核 2GB，最小的资源限额不能超过 0.5 核 100MB。

与 PodNodeSelector 不同，LimitRange 更像是一个兜底手段。因为在代码审查中，对于没有声明 Resources Request 与 Limit 或声明得不恰当的 Kubernetes 资源文件，肯定是要打回的。但人难免会犯错，LimitRange 则是一道“双保险”，保证了所有的 Workload 的资源声明都会在一个合理的范围内。

还有一个话题是如何设置恰当的 LimitRange。首先，Kubernetes 有三种 QoS（Quality of Service，服务质量）。

- Guaranteed：资源的 Request 等于 Limit，在该服务质量下，Kubernetes 会保证分配给对应的 Pod 这么多资源。
- Burstable：资源的 Request 小于 Limit，在该服务质量下，Kubernetes 会保证分配给对应的 Pod 申请的最小资源，当节点资源有空余时，也可以满足 Limit 的资源。
- BestEffort：不声明资源的 Request 与 Limit，在该服务质量下，Kubernetes 对于 Pod 资源没有任何保证，但只要节点资源有空余，就可以全部分配给 Pod。

Kubernetes 设计这三种 QoS，主要意图是提高资源使用率，把服务器的空闲资源给 Burstable 和 BestEffort 的 Pod 使用。而 Kubernetes 本身对于这三种类型的资源优先级也是逐渐降低的。当资源吃紧时，会优先压缩 BestEffort 的 Pod 的资源，甚至驱逐对应的 Pod。因此，全部使用 Guaranteed 事实上违背了 Kubernetes 对于 QoS 的设计初衷。我们的实战经验是对于微服务等 long-running task 使用 Guaranteed，使 Request 与 Limit 相当，保证服务有足够的资源稳定运行。而对于 CI Job 或离线计算任务，则可以使用 Busrtable 或 BestEffort 提高资源利用率。与机器池隔离一致，针对具体场景，在稳定性与资源利用率间做取舍。

4.3.2 Kubernetes 的认证与授权

酷家乐容器化第二阶段的目标是借助 Kubernetes 实现容器的编排调度，提升效率、降低成本。早期业务接入数量不多，并且都是些对 Kubernetes 有一定研究的工程师，大家使用一个共享的凭证访问集群。随着业务和用户数量的增长，这种共享凭证的方式显然不再适合，我们希望每位工程师都有单独的凭证，仅操作自己有权限的集群资源。CMDB 作为 DevOps 各子系统资源配置的基础和核心，与 CI/CD、日志和监控等系统紧密结合。Kubernetes 要接入公司的 DevOps 流程，必然要完成与 CMDB 的打通。接下来主要介绍我们在 Kubernetes 的权限管理的实践。

1. Kubernetes 的用户与认证

Kubernetes 的用户分为服务账户（Service Accounts）和普通账户（Users Accounts）两种类型。服务账户与 Namespace 绑定，关联一套凭证，存储在 Secret 中，Pod 创建时挂载 Secret，从而允许与 APIServer 之间调用。Kubernetes 中没有代表普通账户的对象，这类账户默认由外部服务独立管理（比如 keystone）。

如图 4-3 所示，API 请求要经过多阶段的访问控制才会被接受处理，包括认证（Authentication）、授权（Authorization）和准入控制（Admission Control）等。通俗地讲，认证是验证用户身份，解决用户是谁的问题；授权是限制资源的操作，解决用户能做什么的问题；准入控制在授权后做进一步验证，或者对资源执行默认操作。Kubernetes 支持多种认证方法和授权策略，认证授权完全分离，开发者可以自由地选择合适的方法策略。

现在支持的认证方法主要有 X509 客户证书、静态 Token 文件、Bearer Token、OpenID Connect Tokens、Webhook Token 和认证代理等。支持的授权策略有阻止所有请求（AlwaysDeny）、允许所有请求（AlwaysAllow）、基于属性的访问控制（ABAC）和基于角色的访问控制（RBAC）等。本文不会一一介绍这些方法的所有细节，而是重点讲解酷家乐在 Kubernetes 认证授权中涉及的技术。

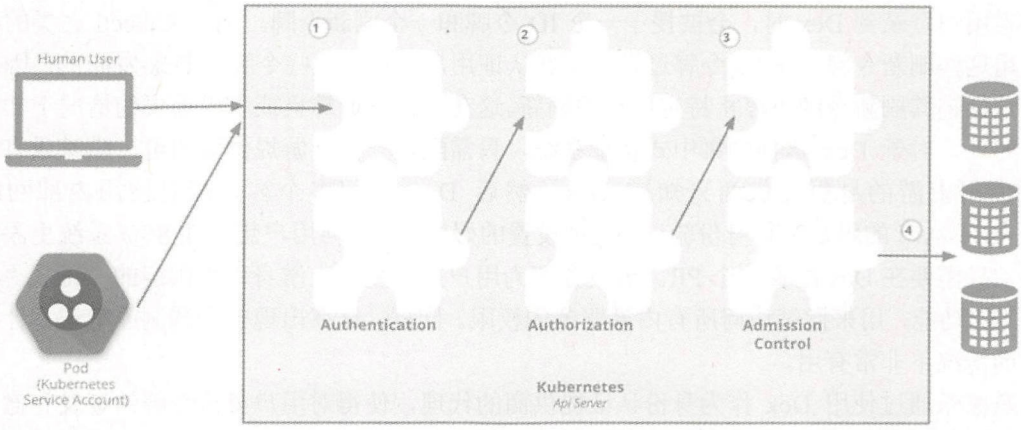


图 4-3　Kubernetes 的用户与认证

酷家乐内部使用 LDAP 做统一身份认证，希望 Kubernetes 也能对接这套系统，而不是建立单独的用户体系。通过调研，确定使用 OpenID Connect 的认证方案，基于 DEX+LDAP 做 Kubernetes 的第三方登录。当然，如果仅想测试的话，也可以选择 Google、Microsoft 等公共身份提供商。

2. OpenID Connect

OpenID Connect（OIDC）在OAuth2[1]上构建了一个身份层，是一个基于OAuth2 协议的身份认证标准协议。OAuth2 本身是一个授权协议，它无法提供完善的身份认证功能，OIDC使用 OAuth2 的授权服务器来为第三方客户端提供用户的身份认证，并把对应的身份认证信息传递给客户端。

OIDC借鉴了OpenID的身份标识，OAuth2 的授权和JSON Web Token[2]包装数据的方式。OAuth2 提供了Access Token解决授权第三方客户端访问受保护资源的问题；OIDC在此基础上提供了ID Token解决第三方客户端标志用户身份认证的问题。OIDC的核心是在OAuth2 的授权流程中，一并给第三方客户端提供用户的身份认证信息（ID Token），ID Token使用JWT格式包装，得益于JWT（JSON Web Token）的自包含性、紧凑性及防篡改机制，使得ID Token可以安全地传递给第三方客户端程序并且容易被验证。

3. Dex 是什么

Dex 是 CoreOS 的一款开源产品，它是一个基于 OpenID Connect（OIDC）核心标准实现的身份认证服务，具有安全、语言与平台无关、支持多种认证后端的特性。Dex 支持从 GitHub、GitLab、SAML、LDAP 和 Microsoft 等多个身份提供商获取用户信息。可以利用 Dex 把 Kubernetes 与现有的用户管理系统进行集成。

在用户身份认证的过程中，Dex 是 Kubernetes ID 令牌的提供商和颁发者，但 Dex 本身并没有身份认证的功能，它仅充当中间人的角色，通过配置后端身份提供商（比如 LDAP）的方式，来提供用户身份认证。此外，Dex 有完善的令牌管理机制，它可以控制 ID 令牌的生命周期。利用 Dex 可以完成 ID 令牌的发布、撤销、刷新和重新认证等操作。

1　OAuth2 是一种授权标准框架，用来解决第三方服务在无须用户提供账号密码的情况下访问用户的私有资源的一套流程规范。

2　JSON Web Token（JWT，包括 header.payload.signature 三部分），是为了在网络应用环境间传递声明而执行的一种基于 JSON 的开放标准。

当用户登录到 Dex 时，会被授予一个 ID 令牌和一个刷新令牌。诸如 kubectl 之类的程序可以用这些刷新令牌，在 ID 令牌过期时重新认证用户。由于这些令牌是 Dex 发布的，因此可以通过撤销其刷新令牌来停止特定用户的刷新。这在笔记本计算机或手机丢失的情况下非常有用。此外，对于 Dex 这样的集中式认证系统，只需配置一次上游提供商即可。酷家乐有一个设置使得上游的身份提供商只知道 Dex，然后 Dex 利用多个客户端对使用内部网站和 Kubernetes API 的用户进行身份验证。这种设置的好处是，任何用户想要往 SSO 系统里添加新服务，只需要在 Dex 配置一个 PR。该设置还为用户提供基于上游身份提供商的一键式“撤销访问”的功能，用来撤销访问所有内部服务的权限。同样，这在出现安全漏洞或笔记本计算机丢失的情况下非常有用。

酷家乐通过使用 Dex 作为身份认证提供商的代理，使得对用户身份令牌的签发和撤销变得精准可控。重要的是，工程师可以使用 LDAP 账号登录 Kubernetes，和内部的身份认证系统保持一致。如图 4-4 所示为交互流程示意图。

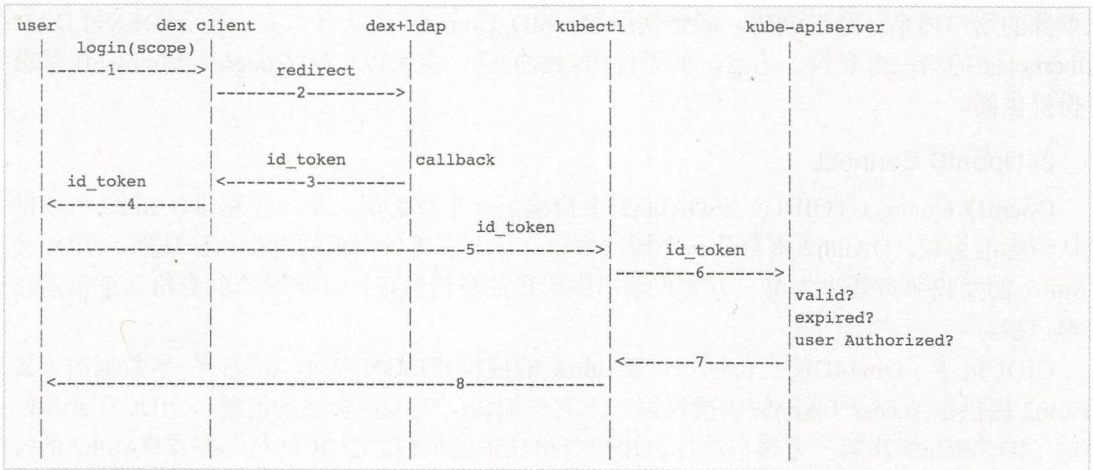


图 4-4　交互流程示意图

1）用户在浏览器发起登录请求。

2）Dex-Client 把请求重定向给 Dex-Server，选择使用 OpenLDAP 的登录方式，Dex-Server 会通过 LDAP 进行身份认证，这时用户就会跳转到授权页面允许访问哪些资源。

3）Dex-Server 把对应的身份认证信息加密，通过回调 URL 传送给 Dex-Client。

4）浏览器中拿到 Token。

5）把 Token 加到 kube-config 配置文件中，让 kubectl 可以使用。

6）kubectl 把 Token 传给 kube-apiserver，kube-apiserver 有 Dex-Server 的公钥可以解析 Token，拿到 username，校验是否过期，看授权是否允许执行该动作。

7）kube-apiserver 把执行结果返回给 kubectl。

获取和安装 Dex 非常简单，Dex 可以直接部署在 Kubernetes 集群中，它需要使用 TLS 证书对通信进行加密。在 Dex 项目源码中，有证书生成脚本和 Kubernetes 的 YAML 配置文件。注意，在 gencert.sh 脚本中把自己的 IP 加进去，configmap 中定义 connectors 类型，此处使用的是 LDAP。

Kubernetes 的配置主要在 kube-apiserver 部分，Kubernetes 的配置参数如表 4-1 所示。

表 4-1　Kubernetes 的配置参数

Parameter	说明	示例
--oidc-issuer-url	公共签名密钥提供者，只接受 https://	https://dex.example.com
--oidc-client-id	颁发 Token 的客户端 ID	kubernetes
--oidc-username-claim	JWT 中 username 的声明	email/name
--oidc-groups-claim	JWT 中 GROUP 的声明	groups

Kubernetes Dashboard 是官方支持的前端管理平台，以上流程可以解决使用 kubectl 访问 Kubernetes 的认证问题，但是无法满足 Dashboard 的认证需求。在 2017 年，Kubernetes 社区就提出要让 Dashboard 支持 OpenID Connect 的认证方式，但至今没有发布。为了让广大工程师可以更方便地访问 Kubernetes 资源，酷家乐开发了 Dex 的一个代理程序（Dex-proxy）。简单来说，用户通过 Dex-proxy 访问 kube-Dashboard。Dex-proxy 会检查 HTTP 请求 Header 中是否包含 Authorization，如果没有则跳转到 Dex-server 的登录授权界面，授权成功 Dex-proxy 会把用户 Token 设置为 HTTP 请求头部继续请求 Dashboard。该 Token 使用 JWT 格式包装，声明了身份提供商、客户端 ID、用户名和有效期等信息。

4. 授权

上述工作完成了用户身份的验证，但是没有解决用户是否有权限访问这些资源的问题。CMDB（Configuration Management Database，中心化配置管理数据库）的每个产品线对应 Kubernetes 一个 Namespace，各个产品线都定义了 owner、appops、developer 和 tester 四种用户角色。CMDB 中的角色信息与 Kubernetes 的 RBAC 访问控制结合使用。每个产品线（Namespace）会在 Kubernetes 中创建应用的四个角色（Role），并且和具体的用户做绑定（Role Binding）。

4.3.3　CMDB 改造

CMDB 以资源管理为核心、融合流程，并以图形拓扑呈现的基础运维平台，是自动化运维平台各子系统共享资源配置信息的基础和核心。在酷家乐容器化实践过程中，把 Kubernetes、Harbor 和 CMDB 打通，实现了其与公司自动化运维子系统的对接。

CMDB 记录了业务运维所需的逻辑信息，提供一个基于服务树和权限绑定的管理模型。基于这一模型，Kubernetes 中的服务与 CMDB 服务一一映射，并且使用 Dex 实现了 Kubernetes 的认证授权管理。此外，根据 CMDB 信息配置服务的机器池、CPU 和内存的资源申请等。根据 CMDB 中的服务重要程度对应不同的应用等级，比如核心、非核心等。基于以往监控数据，把服务划分为 CPU 密集型、I/O 密集型等不同的类型。有了 CMDB 的数据支撑，可以在 Kubernetes 中做更好的调度和资源分配。

CMDB 中不同的应用等级对应 Kubernetes 中不同的 QoS 策略：核心应用对应 Guaranteed，非核心应用对应 Burstable，离线任务等对应 BestEffort。当节点的内存等不可压缩资源耗尽时，优先杀死 BestEffort 类型的 Pods，其次是 Burstable Pods。通过设置服务的 QoS 等级，保证了核心服务质量，同时尽可能多地分配资源给非核心服务。

Kubernetes 支持 Node 亲和性（NodeAffinity）、Pod 亲和性（PodAffinity）以及 Pod 反亲和性（PodAntiAffinity）三种运行时调度策略。Node 亲和性主要解决 Pod 可以部署在哪些 Node 的问题，处理的是 Pod 和 Node 之间的关系。Pod 的亲和性调度主要解决哪些 Pod 可以部署在同一个拓扑域的问题，处理的是 Pod 间的部署关系。基于 CMDB 信息给服务标注 CPU 密集型、I/O 密集型等标签，同时利用 Kubernetes 的 Pod 亲和性调度，很轻松地把相同资源类型的服务

尽量分散调度。这样可以合理分配 Node 资源，提升资源利用率，尽量减少服务间的资源争抢。

为了保证服务质量管理和资源调度机制的有效实行。酷家乐自研了 Admission Controller，在服务创建之初就会做相应校验。Controller 根据 CMDB 信息检查资源 QoS 配置和亲和性调度策略，如果配置为空则会自动填写，配置错误则直接返回失败。

Docker 容器应用的开发和运行离不开可靠的镜像管理，虽然 Docker 官方也提供了公共的镜像仓库，但是从安全和效率等方面考虑，部署私有环境内的 Registry 是非常必要的。Harbor 是由 VMware 公司开源的企业级的 Docker Registry 管理项目，它包括权限管理（RBAC）、LDAP、日志审核、管理界面、自我注册、镜像复制和中文支持等功能。酷家乐内部 Harbor 同样和 CMDB 打通，CMDB 里的产品线对应 Harbor 中的一个项目，项目成员与 CMDB 产品线人员信息一致。

通过上述工作，完成了 Kubernetes、Harbor 等系统与 CMDB 的有机结合。基于 LDAP、CMDB 和 RBAC 机制实现二者的认证授权。利用 CMDB 中长期积累的应用运维数据，更加合理地进行 Pod 调度，保证服务质量，提升资源利用率。

4.3.4 Kubernetes 的包管理工具 Helm

解决了资源隔离与限额问题后，开发人员开始抱怨了：“为什么要写这么长的 YAML 文件，而且其中还有很多信息我完全不关心啊！”

确实如此，Kubernetes 的 API 很完善，各类资源都有大量可配置的字段。但到了酷家乐的场景下，开发人员对于大多数字段，乃至于整个 YAML 文件的语法是否正确，是完全不关心的。开发人员真正关心的是：我要部署哪个镜像？这个镜像起几个实例（Pod）？这个镜像暴露什么端口……

而事实情况是，开发需要复制 Kubernetes 维护人员制订的一份 YAML 模板，修改其中的一些字段，再提交这份 YAML。这种方式难以维护且容易出错，开发人员和运维人员也无法做到 Separate of Concern（关注点分离）。为此，我们引入了 Helm 简化 Kubernetes 应用的部署，实现开发人员和运维人员的关注点分离。

4.3.5 存储方案

PersistentVolume（PV）和 PersistentVolumeClaim（PVC）提供了方便的持久化卷：PV 提供网络存储资源，而 PVC 请求存储资源。这样，设置持久化的工作流包括配置底层文件系统或者云数据卷、创建持久性数据卷、最后创建 PVC 将 Pod 与数据卷关联起来。PV 和 PVC 可以将 Pod 和数据卷解耦，Pod 不需要知道确切的文件系统或者支持它的持久化引擎。PV 是集群之中的一块网络存储。跟 Node 一样，也是集群的资源。PV 跟 Volume（卷）类似，不过会有独立于 Pod 的生命周期。

Kubernetes 中的 PV 支持静态配置以及动态配置，动态卷配置（Dynamic Provisioning）可以根据需要动态地创建存储卷。对于静态配置方式，集群管理员必须手动调用云或存储服务提供商的接口来配置新的、固定大小的 Image 存储卷，然后创建 PV 对象以在 Kubernetes 中请求分配使用它们。通过动态卷配置，能自动化地完成以上两步骤，它无须集群管理员预先配置存储资源，而是使用 StorageClass 对象指定的供应商来动态配置存储资源。酷家乐使用 Ceph RBD 为 Kubernetes 集群提供存储卷。

4.3.6 网络方案

Flannel 是 CoreOS 开源的一个网络项目，目前被使用在 Kubernetes 中，用于解决 Pod 间

直接跨主机的通信问题。它的主要思路是：预先规划出一个网段，每个主机使用其中一个子网，然后每个 Pod 被分配不同的 IP；让所有的 Pod 认为大家在同一个直连的网络，底层通过 UDP/VxLAN 等进行报文的封装和转发。酷家乐目前的 Kubernetes 集群采用了 Flannel vxlan 的网络方案，该方案装了 Flannel 的 Node 节点可以和所有的 Pod 的 IP 直接通信，但是对于非 Kubernetes 集群里面的服务器，无法和 Pod 直接通信。会导致一些服务的使用和调用外部依赖服务时存在问题，不利于容器化的推进。

经过调研，酷家乐使用一个比较简单的解决方案，实现 Kubernetes 集群内 Pod 和非集群服务器的网络打通。方案原理为在 Node 节点上开启 ip_forward，把 Node 节点当成路由器使用，每个 Node 节点转发自己负责的 Pod 的网络，然后在核心交换机上配置静态路由，让请求到 Pod 网络段（比如 172.30.0.0/16）的路由通过各自的 Node 节点转发，这样就能使非 Kubernetes 集群内服务与 Pod 之间进行网络通信。同时，该方案还有高可用加负载均衡的特性，当一台 Node 节点出现故障，只会影响该 Node 节点负责的 Pod 网络，对其他 Node 节点的网络不存在影响，每个 Node 节点会分担流量请求，实现负载均衡功能。

4.3.7　日志与监控

任何生产环境的系统都需要有监控，酷家乐在容器化之前就有内部的监控系统做日志、指标和分布式链路的监控。这些数据的最主要来源都是日志，应用迁移到容器后的监控数据收集方案是需要考虑的。此外，Kubernetes 本身是一个复杂的分布式系统，其本身的监控也是一个需要重点考虑的问题。

本节将讨论如何做 Kubernetes 本身的监控和应用监控数据的采集。

1. 监控 Kubernetes

Kubernetes 的监控方案有很多，对 Kubernetes 的监控需求主要有以下几点：

- 有丰富的监控数据，用于监控集群健康状态和排查问题；
- 系统层面的监控以便做整体的容量规划；
- 提供警报和通知功能。

在具体方案的选择上，酷家乐最终决定采用 Prometheus 组合 Grafana 的方案。Prometheus 本身是一个专门为指标监控而设计的时序数据库，并且附带一个专门处理警报的组件 Alertmanager。配合 Grafana 做数据展示，可以满足对 Kubernetes 本身的监控需求。负责维护 Kubernetes 的工程师通过这套监控方案能够及时地掌握集群的整体状态，发现问题时也能立刻查看细节，帮助找到问题原因，维持整个集群的健康稳定。

2. 数据收集

监控数据的收集方式主要有以下三种：

1）应用直接将数据发往监控系统的收集服务或消息队列。

2）应用将数据发往一个本地的 Agent。

3）应用将数据以日志形式写到磁盘，用一个本地的 Agent 实时地读取日志。

酷家乐采用的是第三种方式，这种方式将文件系统作为一个稳定的数据缓存，可以很好地保证数据的完整性。当 Agent 重启或其他原因导致数据丢失时，可以简单地从之前断掉的点重新读取日志内容。不过这种方式需要占用一定的磁盘 I/O 和空间，一般说来需要给虚拟机单独挂载一个日志盘。

当应用迁移到 Kubernetes 后，依旧沿用这样的方式。一个比较容易想到的改造方案是，

为每个应用的 Pod 添加一个收集日志的 Container，和应用的 Container 共享一个存储日志的 Volume。不过这种方法需要运行非常多的 Agent，有不少的性能开销。酷家乐最终采用的方案是：

- 将所有 Pod 的日志写到 Node 的磁盘上，以 Pod Name 编码的目录路径，并且这个目录在一个单独挂载的日志盘下。
- 用 DaemonSet 部署数据收集的 Agent，读取日志盘下的数据。

通过类似以下的配置挂载应用的日志目录。

```
containers:
- volumeMounts:
  - mountPath: /logs
    name: logs
    subpath: ${PODNAME}
volumes:
- name: logs
  hostPath:
    path: /data/logs
```

- 上面配置的一个问题是 subpath 的值不能是一个变量，因此对 Kubernetes 这部分的源码进行了一定的改造来支持这个功能。
- 关于之后的数据处理和展示完全是监控系统的主题了，和容器化没有太多的关系，不在这里讨论。

4.3.8 小结

在编排自动化阶段中，还解决了本地调试、线上排障、公有云上的使用等问题。对于整个编排自动化阶段，感触较深的有：

1）应用真正迁移到 Kubernetes 后，带来的变化远比想象的大。第一阶段“应用容器化”的积累在此时起到了非常大的作用，在开发人员熟悉 Docker 后，接触 Kubernetes 时学习曲线平滑了许多。因此，关于迁移 Kubernetes，假如团队对 Docker 都还不熟悉，那么完全可以先将自己的应用 Docker 化，再考虑编排层。

2）关于公有云服务与自建的选择。在公有云几乎包办了一切基础设施的今天，这一点其实可以类推到很多技术上。对于一个中等以上的团队，不进行技术积累完全依赖于公有云是不明智的，选择公有云是从性价比的角度出发，而对于 Kubernetes 这样的底层核心技术要有自己的积累，避免厂商锁定。

3）技术投入要以最终产出为原点，要权衡投入产出比。在迁移 Kubernetes 的过程中，其实可以做很多事，比如说容器隔离、做性能更高的网络插件等。这些在技术上都很炫酷，但目前阶段这些事的产出高于所需的投入。当然，在未来达到一个很大的集群规模时，其中某些事可能就会尤为重要，但不是现在。

4.4 酷家乐的服务网格实践

在酷家乐业务版图中，新建立的国际化业务，由于历史包袱较少，与旧业务、旧系统耦合度较低，因此国际化业务的技术承载体——国际站的技术架构中，在全公司范围内第一次使用了基于 Istio 的服务网格体系。这是酷家乐容器化改造三步走战略中，针对第三步的一次有意义的尝试。作者将从酷家乐国际站技术负责人的角度，讲述服务网格是如何被使用起来的，以及酷家乐团队在使用 Istio 的时候遇到过的问题。在撰写本文时，Istio 的发布版本是 0.7.1，但

在文稿修订的过程当中，Istio 正式发布了 1.0 版本，我们迅速升级并进行了体验，本文内容针对 1.0 版本的变化也进行了相应更新。

Istio 的具体技术架构和细节不是本篇文章的重点，国内外均有不少优秀的文章做了详细解读，各位读者可以查阅。

4.4.1 服务网格的发展现状

目前 Istio 的最新版本为 1.0，正如版本号所揭示的，现在的 Istio 社区认为 1.0 是一个稳定的、可以在生产环境内使用的版本。此版本于 2018 年 7 月 31 日正式发布，无疑将为目前热门的服务网格潮流再添一把火。

同时，随着服务网格概念的火爆，其他几个服务网格框架也逐渐进入人们的视野。在国内，目前已有公司引入服务网格的概念，例如新浪微博在自己的 RPC 框架 Motan 上进一步演化出了 WeiboMesh，它在理念上跟 Istio 殊途同归，做到了服务无感知；而阿里巴巴技术团队也基于 Istio 开发了 SOFA 服务网格框架，并且用 Golang 实现了一个 Envoy 组件，基于阿里巴巴技术团队的理念和实际需求，SOFA 在 Istio 的基础上强化了大规模部署情况下的性能和功能，还对非 Kubernetes 业务应用的接入做了较多的适配。感兴趣的读者可以去相关网站和社区查阅相应文档。

4.4.2 酷家乐技术团队应用 Istio 的范围

酷家乐国际站(coohom.com)是酷家乐新近孵化出来的业务，它的核心目标是针对海外用户提供酷家乐引以为傲的设计工具、设计服务。由于国际化业务有可能采用与酷家乐主站（kujiale.com）不同的业务模式，国际站在搭建的时候需要面对的历史遗留问题非常少，基本上是一个全新的起点。所以在做技术选型的时候，最终选择了纯 Kubernetes+Istio 组合的架构。而目前为止，只有国际站使用了 Istio，其他的业务团队尚未引入。

4.4.3 Istio 的安装

Istio 的官方文档中提供了很清晰的安装文档，但是在安装过程中还是遇到了一些问题。

首先是版本选择问题。之前酷家乐在使用的 Kubernetes 版本为 1.7.4，而为了支持 Sidecar-auto-injection 功能，需要 Kubernetes 的版本为 1.9.0 或以上。所以，在安装 Istio 之前，先行将 Kubernetes 升级为 1.9.6。同时，只有版本为 0.5.0 的 Istio 及更新的版本才能支持 auto-injection。

其次是 APIServer 权限问题。由于 Istio 的 auto-injection 功能需要使用版本为 1.9.0 的 Kubernetes 的 MutatingWebhook 功能，需要在 Kubernetes 的 admission-control 中加入 MutatingAdmissionWebhook 和 ValidatingAdmissionWebhook 权限，才能使 auto-injection 正常工作。

安装完毕之后，检验 sidecar 正常开启的方式也非常简单。

```
kubectl label namespace default istio-injection=enabled   //开启某空间的 injection
kubectl -n default get pods  //重启对应 Pod 之后，应能看到 Pod 的 READY 栏为 2/2
```

4.4.4 通过 Istio 的信息进行全自动化部署

由于 Istio 接管并监控了服务间调用的流量，Istio 很容易详细地描绘出集群内部服务间调用的拓扑关系图，在 Istio 中被称为 Service Graph，如图 4-5 所示。Service Graph 除了给人一个比较直观的交互型的展示外，还有一个重要功能是生成描述拓扑关系的 JSON，可以被其他系统读取。

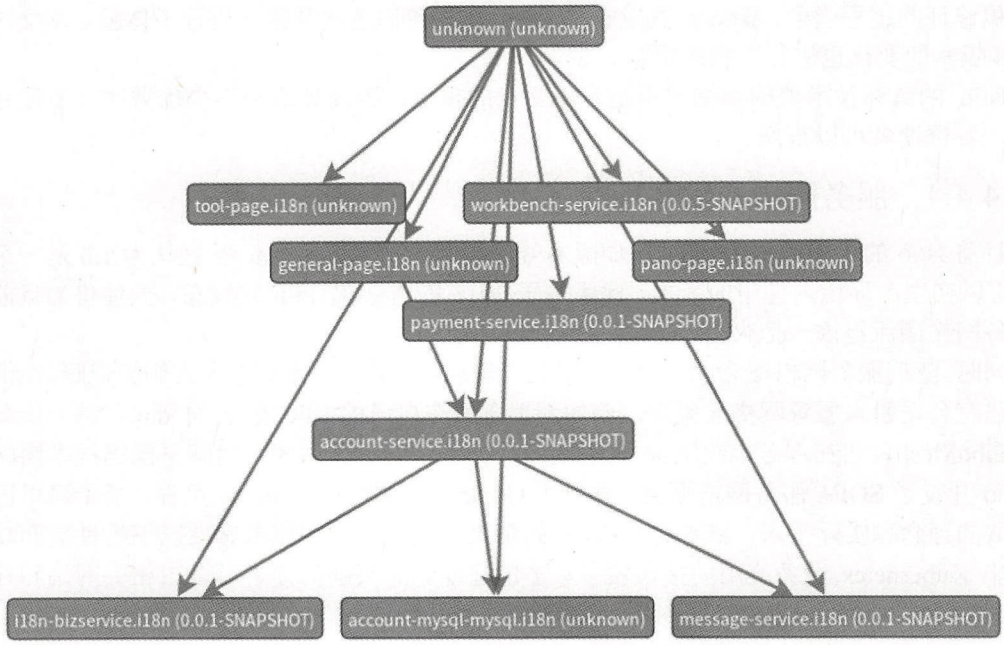


图 4-5　集群内部服务间调用的拓扑关系

根据实际流量生成的拓扑关系，能根据一定规则自动化地生成当前架构下的部署依赖关系，进而描述部署的先后顺序。值得注意的是，这一切都是自动化的，没有人工干预，如图 4-6 所示。

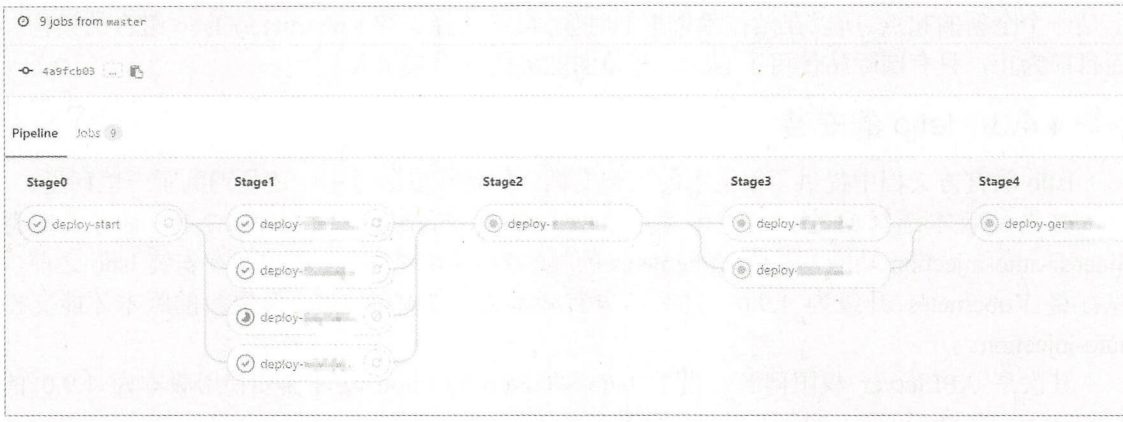


图 4-6　部署依赖关系

在酷家乐国际站的实践中，根据 beta（预发）环境的 Istio 给出的信息，自动拆解出各个服务的部署顺序，然后通过按条件、按顺序地触发 gitlab-ci job，以保证“现有的 beta 环境的服务镜像完全一致地部署到 prod 环境”。通过这一步彻底的自动化，可以很大限度地节省在部署不同环境时的操作成本和监控成本。

这样做基于的假设是“由于预发环境上一定会上线所有接下来会发布的新功能、新改动，而开发人员和测试人员一定会在此环境上充分测到所有的新功能和新改动；根据这个环境的流量生成出来的部署关系图，能反应新的服务间依赖关系”。而如果对于读者所在的技术团队，此假设不成立，那么应该可以根据其他信息制订一个不一样的策略。

通过这项改造，原来需要每个工程师可能会花费几十分钟沟通、监控、配置等工作，压缩

为一个工程师用不到 5 分钟的时间。

4.4.5　通过 Istio + Zipkin + Sleuth 实现调用链路追踪

由于 Istio 可以接管所有的出入流量，这一架构可以很好地实现链路追踪功能：只要 Istio 感知到出入请求，就向某一个调用链查询系统发送信息，调用链查询系统可以做相应处理。这一步对于业务应用是完全透明的，不需要对应用做任何改造。Istio 目前原生支持 jeager 和 Zipkin 收集调用链信息，接入方法直接作为教程文档出现在官方网站上。

而官方文档里没有提到如何用更优雅的方式进行调用链信息的注入（官方给出的 bookinfo 项目示例中，请求追踪所需的 header 是手写代码注入的）。但对于酷家乐基于 Spring Boot 的 Java 应用来说，事情可以变得非常简单：引入 Spring Cloud 的组件 Sleuth。简单来讲，Sleuth 做的事情是：如果一个 HTTP 请求中包含了调用链追踪的 header 信息，就在请求处理进程中将其他的 HTTP 请求也注入相应的 header 信息；如果一个 HTTP 请求中没有相应的 header 信息，则生成它。调用链信息注入逻辑如图 4-7 所示。

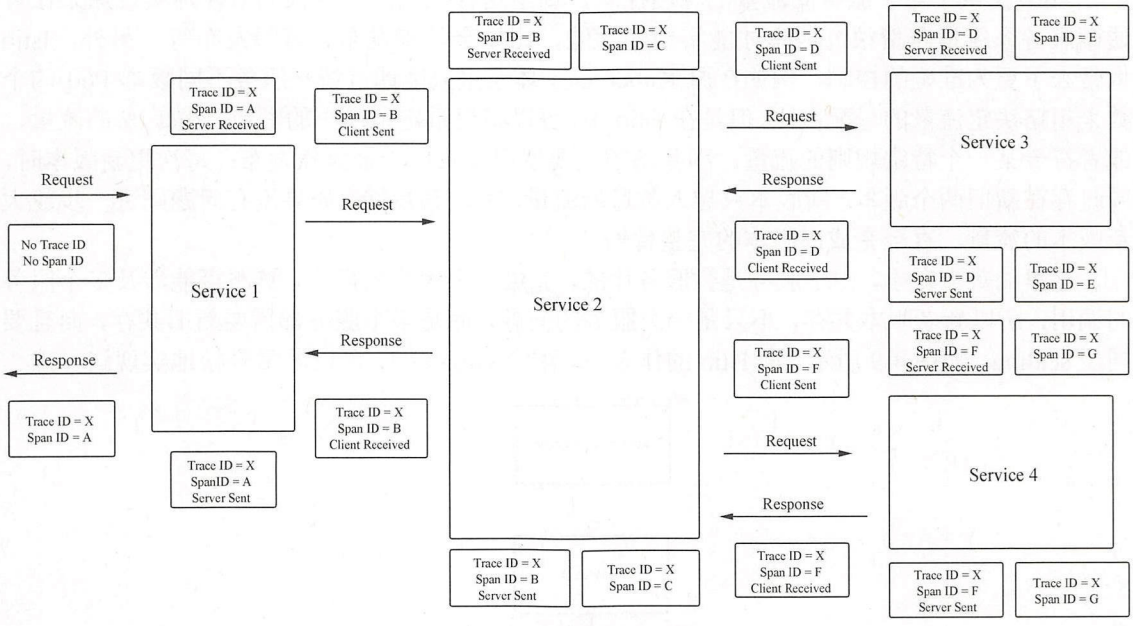


图 4-7　调用链信息注入逻辑

配合 Sring Cloud 的另一个组件 Feign，可以实现不添加任何代码就能完成调用链信息的注入（前提是服务间调用代码都是用 Feign 而不是自建的 RPC 库）。

有了信息注入，下一步自然就是收集，在 Istio 上安装 Zipkin 组件之后，Istio-Proxy 的 Sidecar 会自动收集调用链信息并汇报给 Zipkin 服务器，同样也不需要业务方修改任何代码。

安装方式如下。

```
kubectl apply -f install/kubernetest/addons/zipkin.yaml
```

然后再通过本地的 port-forwarding 查看 Zipkin 的 dashboard。

```
kubectl port-forward -n istio-system $(kubectl get pod -n istio-system -l app=zipkin -o
jsonpath=' {.items[0].metadata.name}' ) 9411:9411
```

此时访问 http://localhost:9411 就可以看到 Zipkin 了，图 4-8 所示为国际站系统中其中一个调用链的展示。

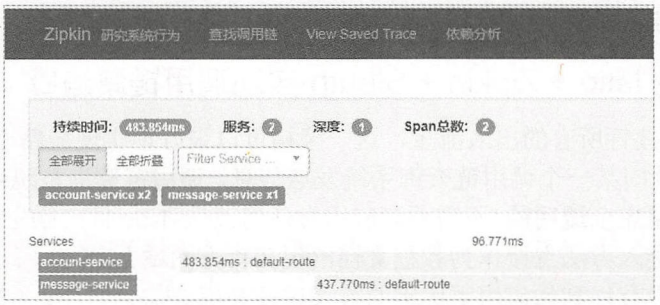


图 4-8　国际站系统中其中一个调用链的展示

通过这种改造，酷家乐用相当低的成本实现了较为全面的链路追踪（前端服务器链路追踪和存储组建的链路追踪需要另外考虑）。这相对于老架构上的链路追踪系统要简易而友好。

4.4.6　通过 Istio 的 routing rule 实现不同的发布策略和版本策略

Istio 提供了对于服务间流量的深度控制，利用这种控制，可以极为方便地实现原来在普通微服务系统里花费很高成本才能实现的功能，比如金丝雀发布、蓝绿发布等。另外，Istio 也带来了更为准确的控制，例如在原 Kubernetes 体系里只能通过统一服务不同版本 Pod 的个数来粗略决定流量的分配占比。但是在 Istio 中，可以将权重定位很小的颗粒，例如 1%的流量，或者符合某一个特定规则的流量。国际站的主要使用场景用于金丝雀发布：每次更新版本时，同时存在新旧两个版本，新版本只导入微量的流量，并观察反馈，如果没有问题则进一步放大新版本的流量，直至完成旧版本的完整替换。

这里的难点在于，由于服务是微服务化的，完成一个用户的操作，链路可能涉及多个服务间调用，所以新老版本共存，不只是一个服务的问题，而是多个服务都需要新旧共存，而且要同步 scaling，如图 4-9 所示。在 Istio 的体系下，配合 Gitlab-CI，可以非常轻松地实现这一点。

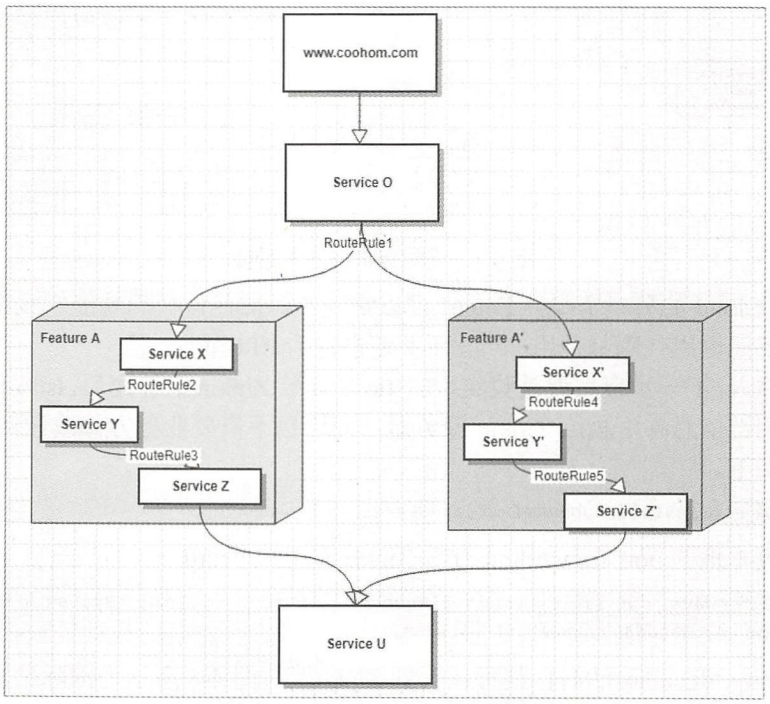


图 4-9　不同版本的服务调用链路

下面是使用的主要步骤：

- 在 Deployment 中，调整 label 中 version 字段的取值方式，它应该直接取服务 repo 中 pom 的版本号，例如 1.0.1，同时，Deployment 自己的名字也要加上此版本号，以免不同的 Deployment 重复。
- Service 需要调整为允许不同 version 的 App，具体为去掉 Selector 中的 version 字段。
- 这里是最重要的一步：需要一个自动化的工具，能够根据应用部署者的输入，自动生成当前所需的部署需要的所有路由规则（RouteRule）。里面需要体现的信息有：需要同时存在几个不同的版本？例如 FeatureA1、FeatureA2、FeatureB1、FeatureB2、FeatureB3。在每个版本中，哪几个服务是需要特殊的路由？它们的版本是什么？随着版本和服务的增多，路由规则肯定会非常多、非常繁杂，靠人工写会大大增加成本和故障率。
- 最后，需要上述生成的路由规则被 Gitlab CI 执行到目标的 Kubernetes 的集群中。酷家乐采用的方式是上述自动生成的路由规则直接作为一个单独的 repo 的代码，它的 Gitlab CI 就是执行这些 rule 的 CI，还可以根据不同的流量规则、流量选取范围，制订不同的 job。如果某一阶段没有问题，则直接执行 CI 的 job 就能进入下一阶段，整个过程中需要人工操作的地方极少。某个部署方案由工程师创建出来供其他人审核，也不需要单独的邮件申请或者系统审核，实际上由相关负责人审核工程师的配置文件改动即可。

上面是酷家乐到目前为止在分版本发布的整体流程，可以看到，这是一种根据 Istio 的能力和架构制订的新流程、新工具。这一点非常重要，只有技术团队能够根据自己的业务情况、Istio 的技术特性，改造原有工作流，改造原有的研发工具，Istio 才有可能较好地落地，发挥最大效果。

经过这种改造，可以省略额外的灰度发布系统、流量管理系统等，同时，路由规则的灵活性和多样性，也给了业务研发团队与业务人员更紧密结合的机会：大功能级别的 A/B 测试可以做了，而且可以完全由业务团队决定例如测试用户、测试版本、测试范围等指标，更好地促进业务迭代。

4.4.7 通过修改 Istio 系统设置实现 Pod 外部访问控制

Istio 中默认会拒绝所有对集群外部的访问，这对于“旧业务进行服务网格改造”场景是不利的。是否能够不改变业务的代码配置，像应用 Istio 之前那样直接调用各种外部服务和组件？答案是可以的。

我们可以通过配置 Istio Proxy 镜像启动的参数，来使 Istio Proxy 选择那一部分 IP 段需要进行流量控制，剩下的流量则会绕开 Istio 直接访问到目的地。具体来说是 includeIpRange 参数。

在 Istio 版本大于 0.4.0 及 Kubernetes 版本大于 1.9 的情况下，Istio 是支持 Sidecar-auto-injection 的，这也是 Istio 可以应用到生产环境的一个必要功能。如果没有这个功能，则需要在每个服务的 Deployment 里注入 Istio Proxy 的配置，才能使得每个服务都有对应的 Proxy 做管理，这样会导致迁移成本和运维成本非常高。

在开启 Sidecar-auto-injection 功能的情况下，只需要调整 Istio-system 里 Configmap 里的配置，就可以实现全局 includeIpRange 配置，对于每个 Pod 来说，重启或者删除 Pod 可以使配置对于该 Pod 生效。具体步骤如下：

```
kubectl -n istio-system edit configmap istio-sidecar-injector
```

在打开的文本文件中找到如下字段。

```
- \"[[ index .ObjectMeta.Annotations \"traffic.sidecar.istio.io/includeOutboundIPRanges\"
\ ]]\"\n    [[ else -]]\n    - \"*\"\n    [[ end -]]\n    - \"-x\"\n
```

将其中的“*”替换为“10. 0. 0. 1/8”，保存并退出，然后删掉并新建所有的 Pod，则此条规则就全面生效了。符合 10.0.0.1/8 网段的 IP 的请求，都会被 Istio 感知到并进行代理，在此网段之外的 IP 请求都会产生直连的效果。

通过这个改动，可以让 Istio 内部服务进行外部调用环节，与无 Istio 的环境中完全一致，这样使得已有业务在迁移到 Istio 环境中时不需要产生额外的改造成本。

4.4.8 Istio 的其他风险

目前版本的 Istio，除前面说的外部访问支持不友好之外，还有如下几个缺点或者风险：

- 性能风险。由于 Istio Proxy 的架构，每次服务间内部调用在原来的服务到服务中间加了两层代理，虽然服务和代理之间的通信并不是网络请求，但这也是一个额外的性能开销。
- 版本风险。当前的 Istio 处于快速迭代中，每个月更新一个大版本，且处于 alpha-stage 和 beta-stage 的 API 随时可能会变更。例如，在 0.7.1 到 0.8.0 的版本中，EgressRule 服务类型的名称就彻底改变了。
- 对于一些技术团队和系统来说，改造成本和学习成本过大。想要利用 Istio 实现较高水平的服务治理，提高研发团队的整体工作效率，就必须让团队内的大多数人都理解并掌握容器、镜像、容器编排和服务网格等概念和工具，并且根据技术上的变化制订新的流程甚至新的组织架构。不同的公司有不同的业务和团队，落地服务网格，需要结合实际的业务需求和团队的技术风格，不要盲目跟风。

4.4.9 小结

作为新一代微服务的架构解决方案，服务网格正在快速走向成熟。经过酷家乐研发团队的落地实战，我们遇到并解决了很多问题，最关键的是减少了很多内部治理工具造轮子，最终降低了系统的维护成本和 DevOps 开发成本。

在实操中，也切实感受到了 Istio 先进的设计理念与活跃的社区。作者提炼的关于 Istio 最有价值的地方有两个：一是 Istio 本身开箱即用，以及各种服务治理相关工具的开箱即用，高度的整合性让服务治理变得简单、让研发流程变得简单；二是 Istio 对“应用无感知”原则的坚持，这进一步明确了 DevOps 和业务研发人员的职责与分工，Istio 配合 Kubernetes，基础设施运维—DevOps—业务研发团队真正实现了各司其职，快速迭代。

作者非常有信心 Istio 将快速成熟并在各大科技企业中落地。同时，酷家乐也在积极探索和实践 Istio 服务网格，也会将相应的进展与经验透过各种渠道发布出来。

4.5 总结

酷家乐在容器化改造中投入了大量的精力，进行了相当多的开发和调整，甚至也根据容器化的进程调整了组织架构和研发流程。

有付出就有回报，酷家乐在改造过程中积累了大量宝贵的经验教训，同时锻炼了整个工程

团队，让工程师们掌握了容器化相关内容的核心概念和基本操作。在此，我们将这些知识和经验分享出来，给其他的技术团队参考。限于作者的能力，文章中难免出现叙述上的疏漏或概念上的偏差，也希望读者们斧正，共同交流、共同进步。

本文由吴叶磊（花名阿磊）、陆兵斌（花名斑斑）、付铖（花名橙子）、牛庆功（花名东悦）共同完成。其中，阿磊完成了第 4.1 节以及第 4.2、第 4.3 节的部分内容；斑斑完成了第 4.2、第 4.3 节的部分内容；东悦完成了第 4.3 章节的部分内容；橙子完成了第 4.4 节内容，以及对全文整体进行润色、总结。感谢他们在写作时付出的努力！

本文的创作由陈东辉（花名子瞻）组织和协调，在此对他的付出表示感谢！

另外，本文的完成离不开酷家乐研发部、Web 后端工程部中间件团队、国际化业务团队、工程效能团队、运维平台团队的大力支持，在此一并致谢。

CHAPTER

5

第5章 个推基于Docker和Kubernetes的微服务实践

个推针对Web服务场景，基于OpenResty和Node.js搭建了微服务框架，提高了开发效率。在微服务的基础上，结合Docker实现了容器化，并采用Consul进行服务注册及发现。后来面对日渐增多的微服务和配置，采用Kubernetes实现了容器编排。

5.1 微服务

5.1.1 微服务简介

如图 5-1 所示，按照传统的单体架构开发方式，所有的服务功能和依赖被打包在一个包中进行发布，以及共享代码和数据。这种模式比较适合规模较小的项目，也有其优点：简单，开发上手比较快；基本没有外部依赖，也没有分布式调用带来的额外开销；以模块方式进行代码复用。

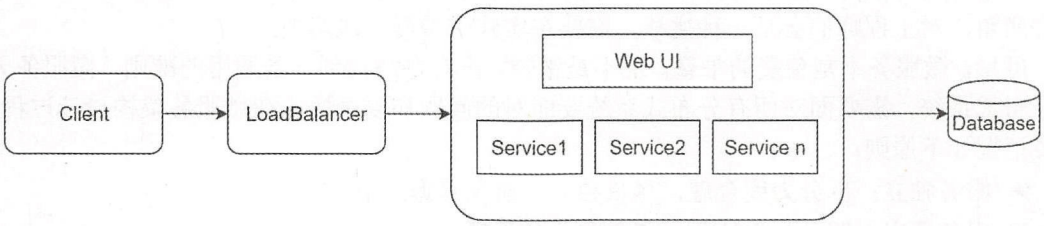


图 5-1 单体架构开发方式

但随着项目推进，功能不断增加，代码库会越来越大，慢慢地会导致产品代码的维护变得越来越困难。这时需要花更多的时间维持模块间的边界，但努力的结果往往无法令人满意。如果遇到新的产品研发需求，为了快速启动项目，团队一般会选择从现有代码中选择一部分模块，组合成新的代码骨架进行新产品的开发。显然，这样做导致多个产品间的代码似曾相识，但又不完全相同。传统的单体架构的缺点为开发成本高、可维护性差、技术选型不灵活、伸缩性差。

微服务架构强调把因相同原因而变化的东西聚合在一起，把因不同原因变化的东西分离开来。微服务架构如图 5-2 所示，特点是：小，专注，一个服务负责一项业务；自治，服务独立部署、升级、扩展；技术异构，服务独立技术选型和开发；服务间松耦合，服务内高内聚；可与组织结构相匹配。

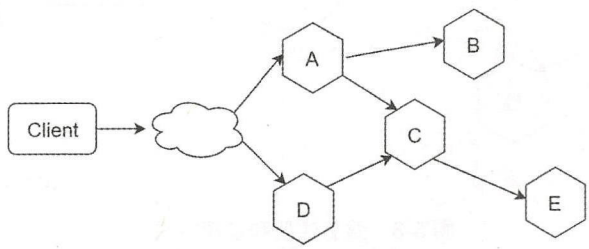


图 5-2 微服务架构

微服务的小，不能简单地通过代码行数衡量，不同语言的表达力不同，不同业务代表的领域对象的复杂度也不同，这些都会影响代码行数。一般来说，合理的微服务架构，团队应该可以在两周内开发完成。这样的微服务更适合小团队开发，也可以避免代码库过大。团队过大或者代码库过大，会出现尾大不掉的尴尬，而现在流行的敏捷项目管理思想，也推荐小团队进行高效迭代。微服务显然是不错的选择。

微服务具有自治性，可以被独立地部署、升级和扩展，微服务对外通过简单 API 调用或者基于事件机制的手段进行相互通信。微服务的自治性带来的独立部署和升级，降低了生产环境变更过程中可能的风险。在特定功能块出现运算、存储等负载瓶颈的时候，相比传统的单体

架构，微服务架构可以独立地进行实例扩展，而不是整体扩容。微服务对语言选择友好，这允许开发者用更低的成本来尝试新的技术。微服务不强制必须采用特定的开发语言，不同的服务可以选择不同的语言实现，比如作为 Web 后端的业务系统可以选择 Node.js 进行开发，更后端可以选择 Java 语言实现支持更高并发或者计算服务。

微服务架构思想要求松耦合和高内聚，这是基础，如果做不到这一点，微服务也就没有价值了。松耦合和高内聚是为了服务可以被独立修改，而不需要该服务的调用方也跟着修改。这要求服务要尽可能少地知道协作的其他服务的细节和信息。在实际的开发中，要做到服务间的解耦并不会很容易，比如一个团队负责开发多个相互协作的服务的情况很常见，这时在修改其中一个服务的时候，工程师需要强迫自己假装并不了解其他服务的细节。这就像电影中一个人分饰两角，对工程师们会是一种挑战，需要在实践中慢慢养成习惯。

可见，微服务不是免费的午餐，更不是银弹，没有办法找到一条通用的准则。微服务天然是分布式系统，需要面对所有分布式系统要面对的问题和复杂性。在微服务架构设计过程中，需要把握如下原则：

- 服务独立：拆分力度合理，尽量独立，避免暴露细节；
- 服务通用：服务要有抽象，避免和业务强耦合；
- 服务无状态：服务需无状态，或通过异步方式解耦；
- 服务松耦合：减少依赖，强调服务层次。

在微服务化过程中，需要将一个产品服务拆分成多个独立微服务，首先要解决如何服务注册和发现的问题，如图 5-3 所示有两种方式可以解决这个问题。

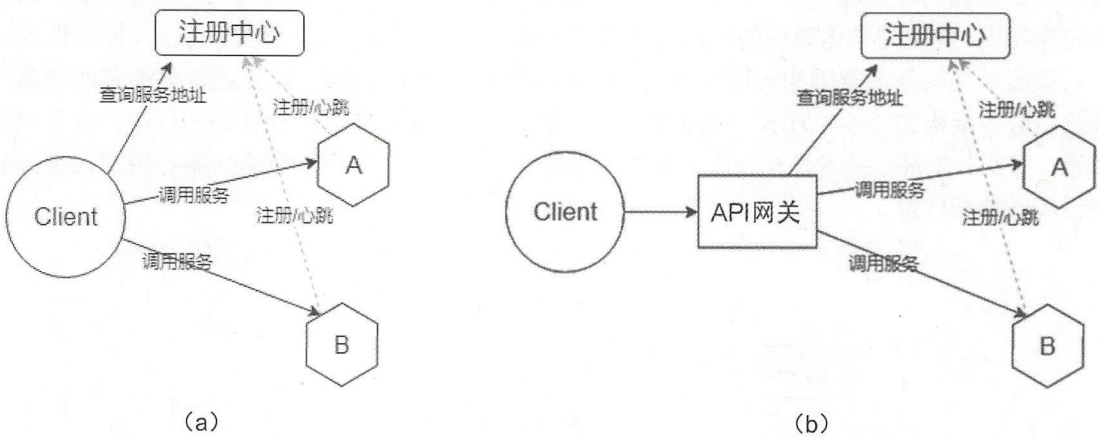


图 5-3　服务注册和发现方式

第一种方式，如图 5-3a 所示，服务提供方向注册中心进行服务注册并直接对外暴露接口，服务调用方通过注册中心发现服务，进而直接调用服务提供方的接口；这种方式扩展灵活，但是客户端维护服务端地址，与服务端耦合程度高，开发成本高，协议升级难。

第二种方式，如图 5-3b 所示，增加服务网关，服务提供方仍然向注册中心注册服务，服务调用方必须通过网关才能访问微服务开放的接口。对于这种方式，服务对客户端透明，可以在网关做统一鉴权、流控，但是需增加 API 网关，还要保证 API 网关高可用，同时增加了额外延时。

在实践中，一般会将两种方式结合使用，外部调用方必须通过网关才能调用服务，而内部微服务之间直接通过注册中心发现服务并进行调用。

5.1.2 微服务实践

个推从 2016 年下半年开始尝试向微服务转型，源于对当时产品业务以及服务场景现状的思考。个推多年来一直专注推送技术服务领域，为广大开发者提供稳定可靠的推送技术服务。同时，个推还深耕大数据研究，开拓了精准营销服务。

在尝试微服务化之前，个推的服务主要由 Java 语言开发，服务不会被拆得很细，由若干个大单体服务共同提供。随着前端逐渐转向 SPA，团队在 Java 语言和前端之间增加了一层很薄的 Node.js 层实现鉴权、协议转换、路由、接口组合等功能。随着服务领域的不断扩展，新的业务的需求也日渐增多和多样，高并发的推送服务场景趋于稳定，Web 服务场景的需求日益增长。团队希望引入时下流行的微服务架构来更好地应对这种需求场景的变化，鼓励尝试新的技术，从而提高开发以及交付效率。

不同的技术栈适合不同的场景。对快速开发、业务逻辑多变、而没有苛刻性能要求的 Web 服务场景，个推尝试选用 TypeScript、Node.js 作为主要开发语言。从一开始就意识到在实践过程中会遇到很多困难，为了少走弯路，决定首先实现一个微服务框架，从而通过框架集中解决微服务过程中最需要解决的问题，比如服务注册发现、服务间的调用、请求鉴权和路由、外部依赖调用等。对于一个产品的开发而言，其复杂度的构成来自很多方面，但是仅有业务逻辑是有必要的，而在普通项目开发过程中，开发者常常会在项目初始化、调用 API 方式的细节、外部资源的适配、库的选择和调试等事务上浪费许多时间。为了满足基于 JavaScript 快速开发微服务的需要，个推内部研发了 WebNode 框架，后面会具体展开介绍。

微服务框架可以用抽屉柜来形容，如果把抽屉柜分为柜体和抽屉，微服务框架就是柜体，规定了接口开放规范，微服务业务代码的分层要求、配置规范等，也提供了服务注册发现、外部依赖的资源化调用等能力，框架代码由团队中有经验的架构师小组负责实现和维护；而微服务业务代码就是抽屉，只需要关注领域业务本身，由具体项目团队成员负责开发。这样，个推把微服务分成了两部分，微服务框架和微服务业务。

用抽屉柜的方式思考服务间相互调用的场景很有意思。一个柜子如果只放一个抽屉，就相当于一个微服务业务独立部署运行；如果放多个抽屉，就相当于多个相关的微服务业务部署在一起作为一个服务提供。如果同一个柜子中有两个抽屉，就是同一个服务实例中的两个微服务业务要相互调用，最简单的方式，通过这个柜子就能够完成，在 Node.js 中，通过 Require 应用规范路径的代码就能实现；而如果是两个不同的柜子中的抽屉，就是两个不同服务实例中的两个微服务业务间需要相互调用，需要通过 RPC 实现。进一步的，如果两个抽屉柜在同一个房间里，也许通过内部服务发现然后通过 RPC 调用可以完成，但如果两个抽屉柜是在两个房间里，就像两个隔离的集群，调用就需要通过专线或 VPN 来保证传输的稳定和安全。更进一步比喻，如果两个抽屉是分别位于两个城市的房子里呢？

可见，微服务间相互调用非常复杂，比如至少有如下几种情况：

- 同一服务实例内，通过进程内调用完成；
- 同一服务集群内，通过服务发现和 RPC 调用；
- 不同服务集群间，通过鉴权后，以 HTTP 等开放接口方式调用；
- 不同 IDC 间，加解密、压缩和鉴权后，经专线或公网，以 HTTP 等开放接口方式调用。

这些情况很普遍地存在，比如服务开发的时候，为了便于调试，多个微服务会被部署在一个服务实例中；上线时，不同服务会被分开部署从而可以独立扩展提供更好的负载；如果服务的一部分需要交付给合作方，这部分服务就会部署到客户的私有机房。

在个推的微服务实践过程中，产生了一些概念：

- 微服务框架，所有微服务的公共外壳，可以看成一个抽屉柜的柜体；
- App，一个微服务的业务代码，可以看成一个抽屉；
- API 调用通道化，微服务间相互调用的统一抽象；
- RAF，微服务依赖的外部服务进行的资源化抽象。

为此，在设计微服务框架的时候，个推提出了 API 调用通道化的概念。微服务化后，会产生大量互相调用的问题。有时多个 App 会被部署在一起，可以通过框架中转实现本地直接调用从而减小开销；有时候特定 App 会被单独拎出来去做 Scale，这时会通过 RPC 调用。但业务代码不关心具体是 HTTP RESTful、gRPC 或者 Thrift 或者甚至是直接对某个本地函数的调用，而关心的仅仅只需要输入输出的数据。因此，通道化在微服务框架里做了一层封装，适配了多种 API 调用的方式。调用的时候，只需要通过指定 api_name，框架就会通过 api_name 在本地配置中找出相应的配置，按照配置选择对应的适配好的调用方式，处理好诸如服务发现、序列化反序列化等细节，完成调用的全部工作。这样，在部署方式发生变化的时候，开发者只需要修改该调用对应的配置，而不需要修改对应的调用接口的代码。

```
api = new_api('my_service_b_content');
res = await api.invoke('methodA', payload);
```

上述代码声明了需要一个名为 my_service_b_content 的 API，随后调用了其 methodA 并拿到了返回值。这个 API 具体位于哪里，则由配置文件决定，不侵入到代码中，该配置由 Consul 下发。可以是一个本地函数的调用，也可以配上一个 URL，使其执行基于 HTTP 的 RPC 调用，甚至可以在鉴权后进行跨 IDC 调用。

同理，外部服务抽象为了资源，也就是 RAF（Rescource Access Framework）。开发业务时不必关心资源的配置工作。以数据库为例，开发人员不关心具体是 MySQL 或 PostgreSQL 或其他持久化实现，关心的只是其资源接口。WebNode 还统一将外部服务 API 化，并统一将参数外置，避免对代码产生侵入，也非常方便使用 Consul 进行参数下发。

```
db = raf.db('my_db');
User = db.define({id: Number, name: String});
user = await User.getById(123);
```

上述代码声明了需要一个名为 my_db 的数据库，然后定义了 User 类型，得到了一个 User 类型的 DAO 对象。随后调用了其 getById 方法查询，代码仅对资源接口依赖。具体数据库的具体选择，都由配置文件决定，不侵入代码中，该配置由 Consul 下发。可以是连一个 MySQL 或是 PostgreSQL，甚至也可以是某个实现了相关接口的代理。

按照微服务思想，完整产品会根据领域模型拆分成多个独立服务，这些服务高内聚，对外提供松耦合的接口给其他服务调用，可以被独立构建和部署。这些服务都使用了非常相似的基础配置，但由于项目的构建时间不同，它们往往非常相似但又略有区别。这使得对于依赖包和配置的管理变得非常碎片化。它们的升级及依赖版本的审计引入不小的工作量。为此，个推在框架统一打包了需要的依赖做了一些统一封装，子项目通过依托框架使用依赖，所有依赖作为框架的子依赖被引入。当需要升级的时候，只要统一升级应用依赖的框架版本，即可升级项目的所有依赖关系。因而每一个业务应用都可以是零配置的，依赖被框架统一管理。

在实现微服务框架过程中，个推的具体技术选型如下：

- 服务发现：通过 Consul 实现，Consul 内建 Service 框架、UI，支持多 IDC、HTTP API，跨语言友好，还用来实现配置中心基础；
- 网关：采用 OpenResty（Nginx+Lua），参考 Orange 插件机制实现，支持灵活的插件配置；

- API 调用：通道化，支持进程内直接调用，同集群 HTTP 调用等方式，数据通过 JSON 格式进行封装，这样的实现简单、可扩展、易开发、测试友好；
- 异步通信：内部自建消息中心，也采用 HTTP 和 JSON，实现基于事件的消息解耦；
- 语言选择：支持 Node.js 和 Java，由于 Nginx 替换为了 OpenResty，也支持 Lua，优选异步非阻塞的脚本语言，上手快，开发成本低，又能实现高并发。

做完技术选型，个推开始对服务进行拆分。首先，进行了更彻底的前后端分离，前端独立出 UI 服务，单独部署；其次，将鉴权、流控、App 流量的路由等做成 API 网关插件；之后，对后端服务进行了分类，分为基础服务和应用服务，把公共服务进行了抽象、抽离，下沉为基础服务，剩余和业务关联紧密的是应用服务。最后，基础服务和应用服务按照领域模型进行了微服务拆分和实现。

如图 5-4 所示，个推开发了 WebLua、WebNode、Javams 框架，框架封装了服务间通信、路由的解析等基础功能，对外提供应用插槽，将服务模块抽象成一个个独立的 App，只需要在框架的应用插槽中插入对应的 App，就可以实现应用的部署。有了这样的框架，开发者只需要关注具体的业务逻辑实现，而不需要考虑服务之间如何通信，如何调用 MySQL、Redis 等存储服务等，极大地提高了开发者的开发效率，减少了出错的可能性。并且 App 可以很方便地实现部署，当然，每个 App 甚至可以独立地作为一个服务进行部署。

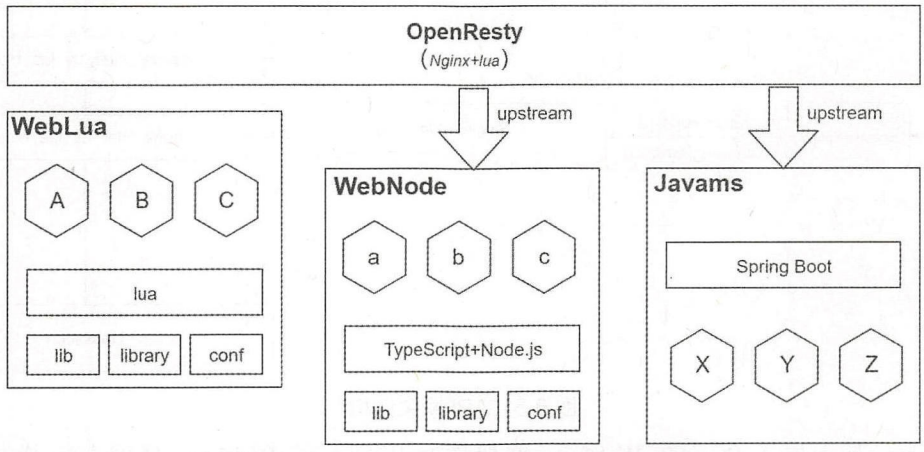


图 5-4　WebLua、WebNode 和 Javams 框架

基于 WebLua 框架主要实现了微服务流量的代理和 API 网关。进入微服务的流量由 OpenResty 统一代理，通过 Lua 语言实现了 dyups 模块，安装于 WebLua 框架之上，根据流量特征实现动态路由。API 网关作为所有流量的统一入口，在微服务架构中扮演着重要的角色。在业务的开发过程中，有许多需求是各产品共有的，如应用服务的安全防范、流量限制、分流、A/B 测试等，这些共性的需求非常适合在 API 网关统一解决，这样可以有效地避免各个产品开发团队做重复性的开发。而对于 API 网关来说，要能够易于扩展这些功能，就需要一个更合理高效的架构，这样即使未来有新的需求，也可以很方便地实现功能的扩展。

在设计 API 网关的实现方案时，也调研了其他成熟的网关产品，如 Kong、Orange 等。Kong 和 Orange 都是很优秀的网关产品，通过插件的机制提供了丰富的功能，并且支持自定义插件对其功能进行扩充，同时还提供了 Web 界面和 API 对网关进行管理和配置插件。也曾考虑过将网关服务迁移到 Kong 或 Orange 上。但是评估过后，Kong 或 Orange 都不是特别适合，它们的所有插件是在 OpenResty 的 init 阶段全局加载的，所有的流量都需要通过所有的插件规则进行过滤，虽然可以通过不同的规则进行不同的处理；而个推的微服务架构中，网关是被多个

产品共同使用的，每个产品关于流量的控制规则都不太一样，不太合适在相同的插件中进行处理，这样的话，规则会非常的复杂。同时，每个产品都配置有不同的对外端口，还需要动态路由到不同的后端服务，如果放在 Kong 或 Orange 中实现起来也不容易。Kong 或 Orange 都依赖外部的 DB 进行存储，而个推的网关主要依赖于 Consul 进行相关配置的更新和存储。个推网关中基于自研的 WebLua 框架实现了很多种 Auth 方式，如果迁移到 Kong 或 Orange 上，都需要重新基于插件的规范进行开发，这也会带来一定的开发量。基于以上的考量，决定将 Orange 的插件机制引入到 API 网关中，这样既可以享有 Orange 的插件机制带来的功能扩充的便利性，又不改变现有的技术栈。API 网关架构如图 5-5 所示。

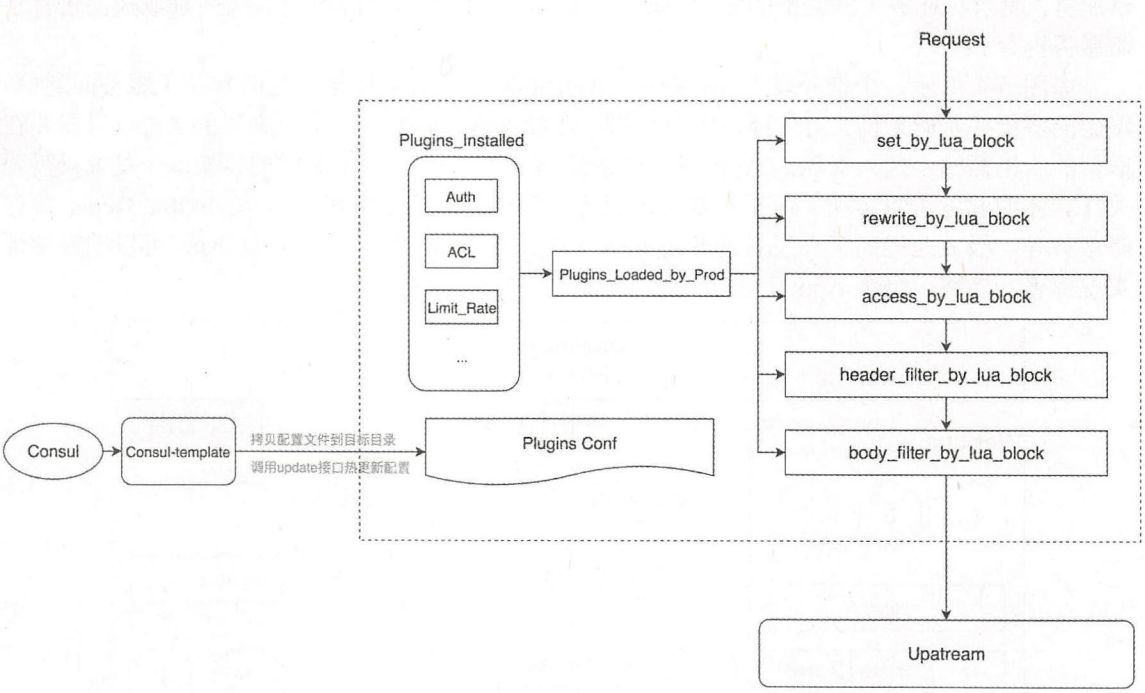


图 5-5　API 网关架构

Apigate 基于以上介绍的 WebLua 框架进行开发，在 WebLua 基础之上开发了名为 weblua-app-apigate 的 App，插件的加载、流量的过滤都在该 App 中实现，该 App 还提供了用于配置热更新的接口。

而 Apigate 的插件也按照 WebLua App 开发规范进行开发，可以方便地安装到 WebLua 框架中并由 Apigate 进行加载。为了降低插件流量规则配置的复杂度，个推的 Apigate 的插件开发设计给了开发者更大的自由度，开发者可以在基本的配置规范下，自定义字段和配置解析方法来实现流量的识别和过滤。

插件的流量规则的配置统一放在 rules 下，rules 是一个数组，每个元素一条规则，每条规则固定有一个 pattern 字段，用于匹配流量，支持正则表达式，continue 字段表示匹配当前插件后，是否继续匹配后续插件，如果为 false，则执行完当前插件的操作后不再匹配后续的插件。规则中还可以添加其他字段，由开发者自行决定其他字段的定义和解析使用方法。

```
{
    rules={
      {
        pattern='/login\\?(.*)$',
```

```
        continue=true|false
      },{
        pattern='/user/info',
        continue=true|false
        }
        ……
    }
}
```

目前个推的 Apigate 已经实现了如下多个插件：

- rewrite_uri：实现对 uri 的重写，类似于 Nginx 中 rewrite 指令的功能，通过正则匹配来重写 uri；
- dyups_svc：根据流量的特征动态修改 upstream；
- auth_session：实现 login、logout 和通过 session 进行鉴权的功能；
- acl_privilege：实现对 uri 的权限校验，网关通过调用后端的 privilege 服务，判断当前用户对当前 uri 是否有权限访问。

如果有新的需求可以通过开发相应的插件实现网关功能的扩展。

WebNode 框架使用 Node.js，最早基于 Express 和 co 实现，个推完成了框架基于 Koa2.0 的升级，而业务相关的 App 大多采用 Node.js 开发，如同 Lua 开发的 App 一样，安装到 WebNode 的应用插槽之上。

Javams 是个推为了针对高并发场景快速构建微服务研发的框架，在 Spring Boot 的基础上，包含 RPC 框架、Trace、缓存、健康检查等组件，一站式提供。

个推建议所有的 App 进行独立的代码版本管理，事实上，不仅只对 App 的代码进行版本控制，随着微服务实践的深入，还使用了 Docker、Kubernetes 等技术，也对 Dockerfile、Kubernetes 的部署脚本，以及配置进行版本控制。这样可以实现可重现、可追溯、可回滚的部署，也是实现 DevOps 的基础。

总之，个推结合微服务的思想，开发了自己的 WebLua、WebNode 和 Javams 框架，对产品线的服务进行拆分，将服务拆成了一个个独立的 App，从而解决了如下问题：

- 统一对接入层和路由层进行处理，开发人员只需要关注业务开发；
- 使得新建、改变甚至重写一个服务的成本非常低；
- 整体系统更具有弹性，可以更好地处理服务不可用和功能降级问题；
- App 之间边界清晰，可以做单可拔插的配置管理，也容易针对需要扩展的服务进行扩展；
- 部署简单，而且风险可控；
- 能够以小团队的形式进行协作，一个小团队使用几个小 App，沟通成本低；
- 开发的微服务可以很方便地复用，快速上线新产品。

5.2 容器化

1. 微服务存在的问题

在微服务化过程中，发现微服务并不是万能的，还有很多痛点没有解决：

- 业务系统 QPS 低的服务特性，为节省资源，会采用混部。近 10 个 App，需要部署近 20 个实例，混部在几台机器上，管理难，隔离不彻底，服务间很容易相互影响，端口

等的设置、管理困难；

- 缺少统一的部署脚本，缺少配置中心，暴露了配置细节给测试和运维；
- 微服务对 App 进行了拆分，服务拆细，在解耦合，复用上存在优势，但也存在服务多、配置复杂的问题，特别是多个微服务间相互依赖，整体提供服务，没有服务组的概念。

在容器出现之前，这些痛点很难解决，微服务实践起来也是一件很难的事情，微服务要运行起来需要一个隔离的环境，而这个环境不能对外部有依赖，同时环境要“微”，不然会造成资源的浪费。在容器出现之前，可以使用虚拟机进行环境的隔离，运行微服务，但是虚拟机的粒度比较大，不够“轻”，资源不能充分利用，并且虚拟机重启也会比较耗时，管理起来也没那么容易。容器的出现提供了一个非常合适的运行环境。

容器是一个轻量级的虚拟环境，资源占用很小，一台服务器可以运行成百上千的容器；直接调用系统内核实现对操作系统的访问，秒级启动；开发人员交付给测试人员和运维人员的是一个可运行的环境，交付出去就可以直接运行，减少了运维的难度，可以做到“Build once，Run anywhere”；容器有很多编排管理工具可用，使得容器的管理非常方便。

2. 容器化实践

个推选择了 Docker 进行容器化实践，首先搭建 Docker 集群环境。选择了 Calico 作为网络插件，Calico 是一套基于路由（BGP）的 SDN，通过路由转发的方式实现容器的跨主机通信。Calico 有如下的特点：

- 纯三层的数据中心网络方案；
- 利用 Linux Kernel 实现了一个高效的 vRouter 来负责数据转发；
- 可以直接利用数据中心的网络结构，不需要额外的 NAT、隧道或者 Overlay Network；
- 提供丰富而灵活的网络 Policy（基于 iptables ）；
- 支持很细致的 ACL 控制；
- 性能高、可控性高、隔离性好。

基于以上的特点，选择 Calico 作为容器集群的网络组件，实现跨主机 Docker 容器之间的通信。

基于 Docker，个推规划了镜像构建和产品服务体系，对于镜像的构建，分为四个级别的镜像：

1）系统级别的镜像。在选择系统镜像的时候，考虑到系统的成熟稳定和版本的更新维护，选择了安全补丁更新及时、包更全的 CentOS7 镜像，在此基础上安装了需要的工具，从而构建出自己的系统镜像；

2）框架基础环境的镜像。在系统镜像的基础上，安装服务框架依赖的软件和语言环境，如 OpenResty、Node.js 及 Java 等；

3）框架级别的镜像。在基础环境镜像的基础上，安装服务框架，构建出框架镜像，以此为基础可以构建出具体的应用服务镜像；

4）应用服务镜像。在框架镜像的基础上，安装应用 App 构建应用服务镜像。

产品服务体系分为四类：API 网关服务；辅助服务，包括服务注册中心、配置中心、消息中心；产品应用服务；产品基础服务。

基于 Docker 的服务框架如图 5-6 所示。

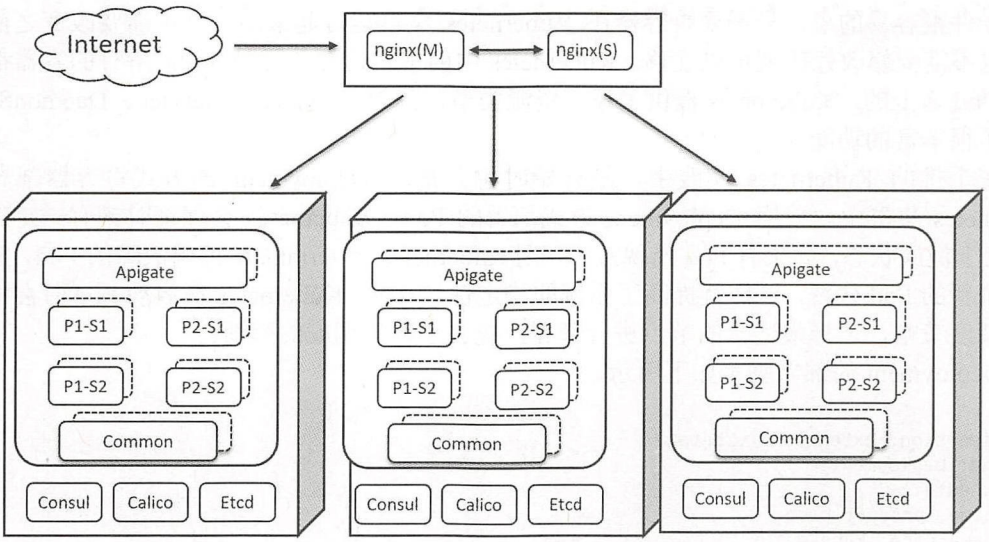


图 5-6　基于 Docker 的服务框架

在服务框架中，Consul 扮演了重要的角色，服务注册发现和配置管理都是基于 Consul 实现的。在选择 Consul 时，也对比了 ZooKeeper 和 etcd，如表 5-1 所示。

表 5-1　Consul、ZooKeeper 和 etcd 对比

特　征	Consul	ZooKeeper	etcd
服务健康检查	服务状态	弱	连接心跳
多数据中心	支持	—	—
一致性	Raft	Paxos	Raft
使用接口	HTTP 和 DNS	客户端	HTTP/gRPC
33Watch 支持	全量/支持长轮询	支持	支持长轮询
安全	acl/HTTPS	acl	支持 HTTPS
自身监控	metrics	—	metrics
UI	有	—	—

通过比较，Consul 和 etcd 都是不错的选择，都提供了与代码解耦的 HTTP 接口方式，但 Consul 更适合现有的需求，且提供了统一的 UI，再加上 Consul-template 更方便地支持了调用形式的解耦。最终选择了 Consul 作为服务注册中心，同时还作为配置管理的基础提供服务。

Docker 的使用解决了微服务实践中的很多痛点，比如微服务之间环境的隔离，容器化后即使多个微服务部署在同一台服务器上，也可以互不影响；微服务的水平扩展，升级也变得非常容易；容器化也使得整个集群的维护变得更容易。但是容器化后也还有一些问题需要解决，主要有：容器的调度，水平扩展都需要人工干预；容器的故障发现和恢复需要通过监控系统或者人工去实现；容器的监控、日志收集分析实现起来还是有一定的难度。

而这些需求其实都可以通过容器编排工具实现，而容器编排工具中的佼佼者就是 Kubernetes，为了使容器集群更便于管理和维护，个推也引入了 Kubernetes 作为编排工具。

5.3　Kubernetes 实践

Kubernetes 作为一个容器的管理工具，不会侵入容器，所以容器化向 Kubernetes 集群做迁

移是一件很容易的事，只需要将容器在 Kubernetes 集群运行起来即可，不需要改变之前的架构，也不需要修改代码就可以迁移。Kubernetes 中的基本运行单元是 Pod，所有的容器都是运行于 Pod 之上的，Kubernetes 提供了很多资源类型，如 Deployment、Service、DaemonSet 等，提供了很丰富的功能。

在个推的 Kubernetes 实践中，刚开始时也只是用 Deployment 的方式将容器部署到了 Kubernetes 集群中。采用 Deployment 方式部署的 Pod，Kubernetes 会保证时刻有一定副本的 Pod 处于健康状态，如果有 Pod 出现故障，则 Kubernetes 会 terminate 掉出故障的 Pod，重新启动一个新的 Pod 代替，这样就提高了系统的稳定性。同时，Kubernetes 在启动 Pod 时会根据特定的调度策略，选择最适合的节点进行部署，免去了人工调度的麻烦。

Deployment 的部署脚本如下所示。

```
--
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: product-name
  namespace: kube-getui
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: product-app
    spec:
      containers:
        - name: product-container
          image: product/image:0.0.0.1
          env:
            - name: NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
          livenessProbe:
            httpGet:
              path: /health
              port: 80
              scheme: HTTP
            initialDelaySeconds: 60
            timeoutSeconds: 5
          readinessProbe:
            httpGet:
              path: /health
              port: 80
              scheme: HTTP
            initialDelaySeconds: 5
            periodSeconds: 5
          volumeMounts:
            - mountPath: /app/data
              name: app-data
            - mountPath: /app/logs
              subPath: app
              name: applog
      volumes:
        - name: app-data
          hostPath:
            path: /app/product/data
        - name: applog
          emptyDir: {}
```

Kubernetes 提供了很多强大的功能，它的很多设计思想与微服务很契合，关于服务的注册和发现，Kubernetes 也提供了一种资源类型——Service。Service 是应用服务的抽象，通过 Labels

为应用提供负载均衡和服务发现。匹配 Labels 的 Pod IP 和端口列表组成 endpoints，由 Kube-Proxy 负责将服务 IP 负载均衡到这些 endpoints 上。Kubernetes 为每一个 Service 分配一个唯一 cluster IP，集群内部可以通过虚拟的 IP 访问服务，所有的请求会被自动均衡到后端的各个 Pod 中。

如图 5-7 所示，Service 会根据定义中的 Label（app=myapp）找到打有对应 Label 的 Pod 作为该 Service 的 endpoints。对 Service 的访问会均衡地将负载部署到所有的 endpoints 上。

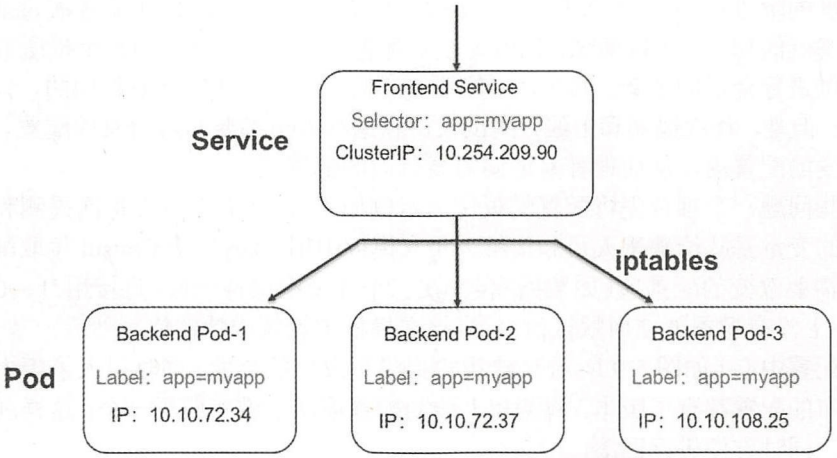


图 5-7 Kubernetes 中 Service 的负载均衡

Kubernetes 可以通过 Service 的 VIP 访问对应 Pod 提供的服务，但是 VIP 却不容易拿到，而 Service 的 Name 是部署 Service 时设定的，如果能够通过 Service Name 直接访问 Service，就会方便很多。Kubernetes 提供了一个插件 Kube-DNS 可以将 Service 注册到 DNS，直接通过 Service 的名字即可访问 Service，推荐安装。Kube-DNS 的工作原理如图 5-8 所示。

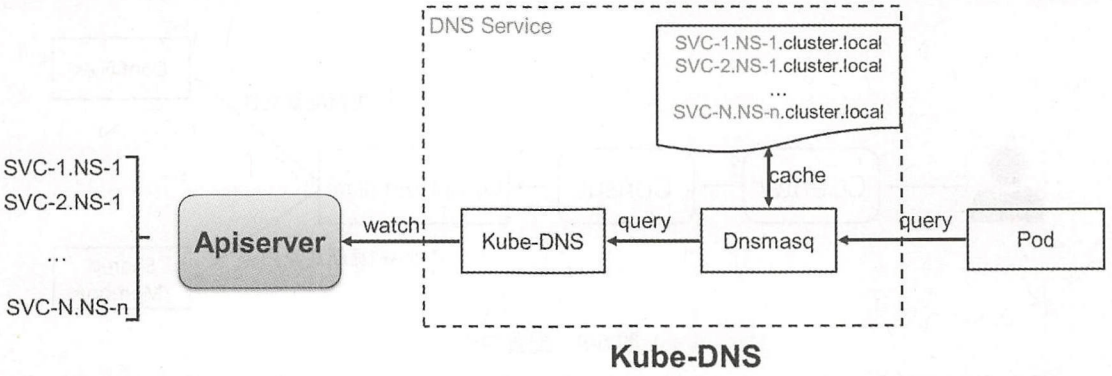


图 5-8 Kube-DNS 的工作原理

基于 Kubernetes 的 Service 和 Kube-DNS，个推重新设计了产品服务的注册与发现机制。所有的服务通过 Kube-DNS 注册到 DNS 上，不再注册到 Consul 上，服务之间的调用直接通过 Service 名字访问，负载均衡直接由 Service 提供，不再由 API 网关实现。由于 Service 的域名只在 Kubernetes 集群内部有效，在集群外部并不能直接访问，为了使 Apiate 能够对外暴露服务，Apigate 的服务注册和发现还是通过 Consul 实现，前置 Nginx 直接从 Consul 上获取 Apigate 的上暴露的 IP 和端口，映射到主机端口上，实现服务的对外暴露。

随着 Consul 作为服务注册中心的角色逐渐被 Kubernetes 的 Service 和 Kube-DNS 替代，对

Consul Agent 的部署方式也进行了调整。之前 Consul Agent 会伴随着服务应用一起运行在同一个容器中，这样才能保证应用服务能够将服务注册到 Consul 上，现在不再需要向 Consul 注册服务了。因此，把 Consul Agent 从应用容器中剥离出来，在 Kubernetes 集群上单独部署服务，所有的应用服务都通过连接 Consul Agent 服务从 Consul 上拉取配置，这样 Consul Agent 只需要以 Deployment 的方式部署几个副本即可，既能节省系统资源，又能够保证服务的高可用。

Consul 除了作为服务注册中心，还有一个重要作用是作为配置中心进行配置的管理。在只有容器集群的阶段，所有的配置都通过 Consul 的 Web UI 进行创建和配置，Consul 的 Web UI 只提供了最基础的功能，比较简陋，同时无法对配置进行版本控制。当时个推所有的配置以配置文件为单位进行全量的更新。其实，虽然环境不同，但大部分配置是相同的，只有少部分配置可能不同。但是，每次提测和上线，测试人员和运维人员都要面对所有的配置，很难聚焦到真正需要改变的配置项，从而很容易遗漏需要修改的配置项。

基于这些问题，个推首先将配置模板化（示例如下），从而可以在每次提测和上线时将真正需要修改的变量暴露给测试人员和运维人员（模板中用‘key’从 Consul 拉取配置，必须配置），其他不需要改变的配置项（如数据库名，在三个环境中保持一致，直接用‘keyOrDefault’，如果 Consul 上没有配置就使用默认值），直接在模板中设置成默认值。然后，基于 Consul 开发了自己的配置中心（如图 5-9 所示），给用户提供更友好的界面。同时引入了版本控制功能，这样每次发布的配置都有了版本，提测和上线的配置项从一堆变成了一个，这样就可以更容易地管理配置，同时方便进行回滚。

```
module.exports = {
    host:'{{ key "/base_path/mysql/host" }}',
    port:{{ key "/base_path/mysql/port" }},
    user:'{{ key "/base_path/mysql/user" }}',
    password:'{{ key "/base_path/mysql/password" }}',
    database:'{{ keyOrDefault "/base_path/my_database" "default_database" }}'
};
```

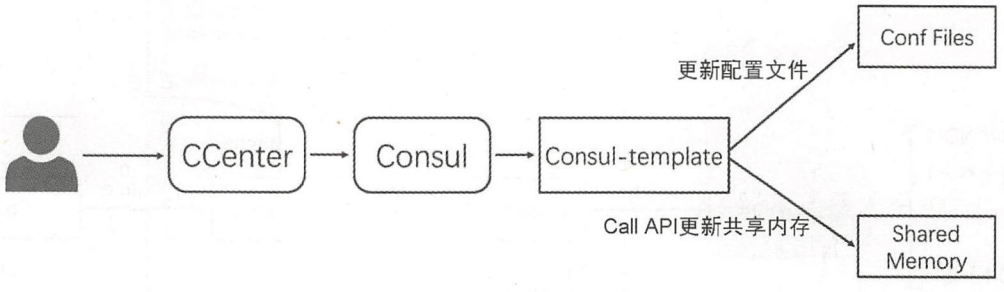


图 5-9　配置中心

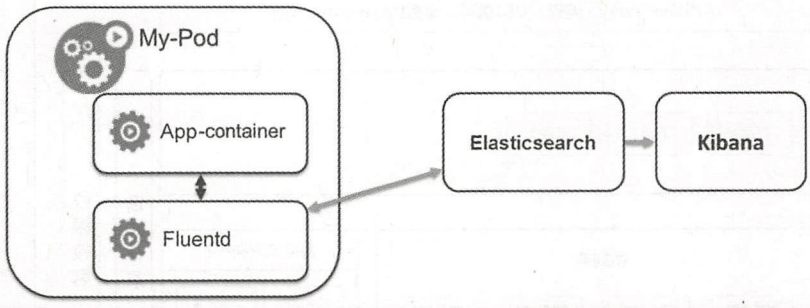
在未引入 Kubernetes 之前，容器和集群的监控和日志处理分析是一件很难的事情，一般需要在每个容器中部署一个 Agent 收集监控指标和日志，然后统一汇报给监控或日志中心进行处理。对于一个没有运维经验的开发者来说，部署这样一套监控或日志处理系统是一件很麻烦的事。Kubernetes 提供了插件机制，可以很方便地扩展自身的功能。而监控和日志处理系统都可以通过插件的形式很容易地部署到 Kubernetes 集群，实现对整个集群的监控和应用 Pod 的日志处理，只需要下载合适版本的部署脚本，然后通过 kubectl 执行部署脚本，待部署的服务运行起来后修改一些配置，即可完成监控系统和日志处理系统的部署。

Kubernetes 中的监控系统可以通过部署 Heapster、InfluxDB、Grafana 三个插件实现，Heapster 是一个收集者，将每个 Node 上的 cAdvisor 的数据进行汇总，然后导入第三方工具（如

InfluxDB），通过 Grafana 进行展示，如图 5-10 所示。

图 5-10　Grafana 监控页面

Kubernetes 中常用的日志处理系统是 EFK 即 Elasticsearch、Fluentd、Kibana，如图 5-11 所示。其中，Elasticsearch 存储日志并提供搜索，Fluentd 负责收集日志，需要部署到应用 Pod 中，但是不会侵入应用容器，Kibana 负责查询和展示日志。



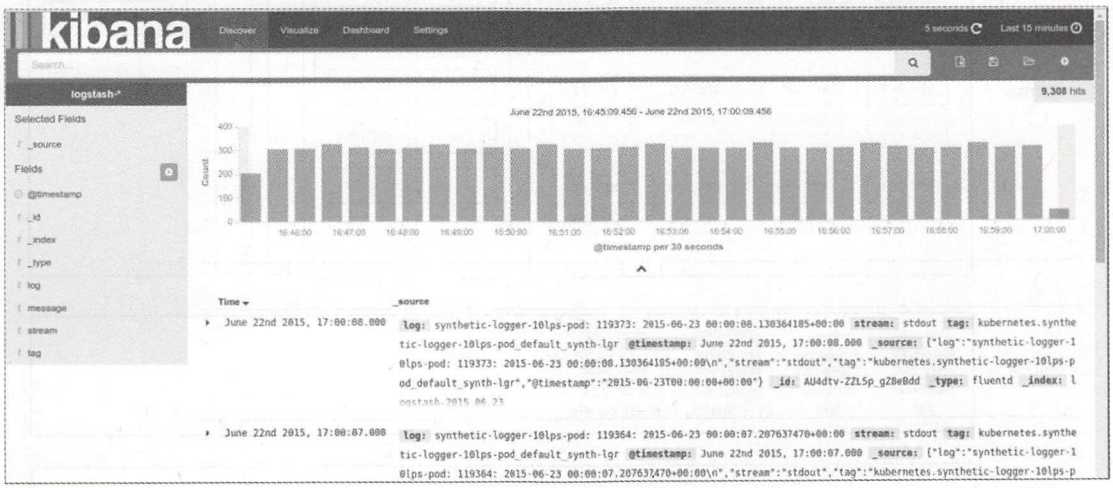


图 5-11　日志处理系统 EFK

为了追踪服务请求的链路，个推还引入了 Zipkin。Zipkin 是一款开源的分布式实时数据追踪系统，是 Twitter 公司基于 Google Dapper 的论文设计开发而来的。其主要功能是聚集来自

各个异构系统的实时监控数据，用来追踪微服务架构下的系统延时问题。

Zipkin 也作为一个插件进行部署，同时还需要通过应用代码修改收集请求链路中的必要信息。

个推基于 Kubernetes 的微服务架构如图 5-12 所示。不同产品线可能部署在不同的 Kubernetes 集群或通过不同的 Namespace 进行隔离，同一条产品线中可能会部署多个产品，每个产品线的服务采用分层架构。

- 前端展示层：面向的是最终的用户，每个产品有独立的前端网站面向用户提供服务。
- API 接口层：流量的统一入口，负责流量的动态路由、分流限流、鉴权、统一权限控制、安全控制等。
- 业务应用系统：负责产品的业务逻辑的具体实现。
- 服务中心：提供了所有的基础服务和与业务相关的服务，还提供了很多基础服务中间件，实现如消息队列、日志管理、配置管理、监控预警等功能。
- 数据存储：主要提供系统依赖的数据存储服务，如 MySQL、Redis、分布式的文件存储等。
- 运行环境：整个服务运行的物理环境。

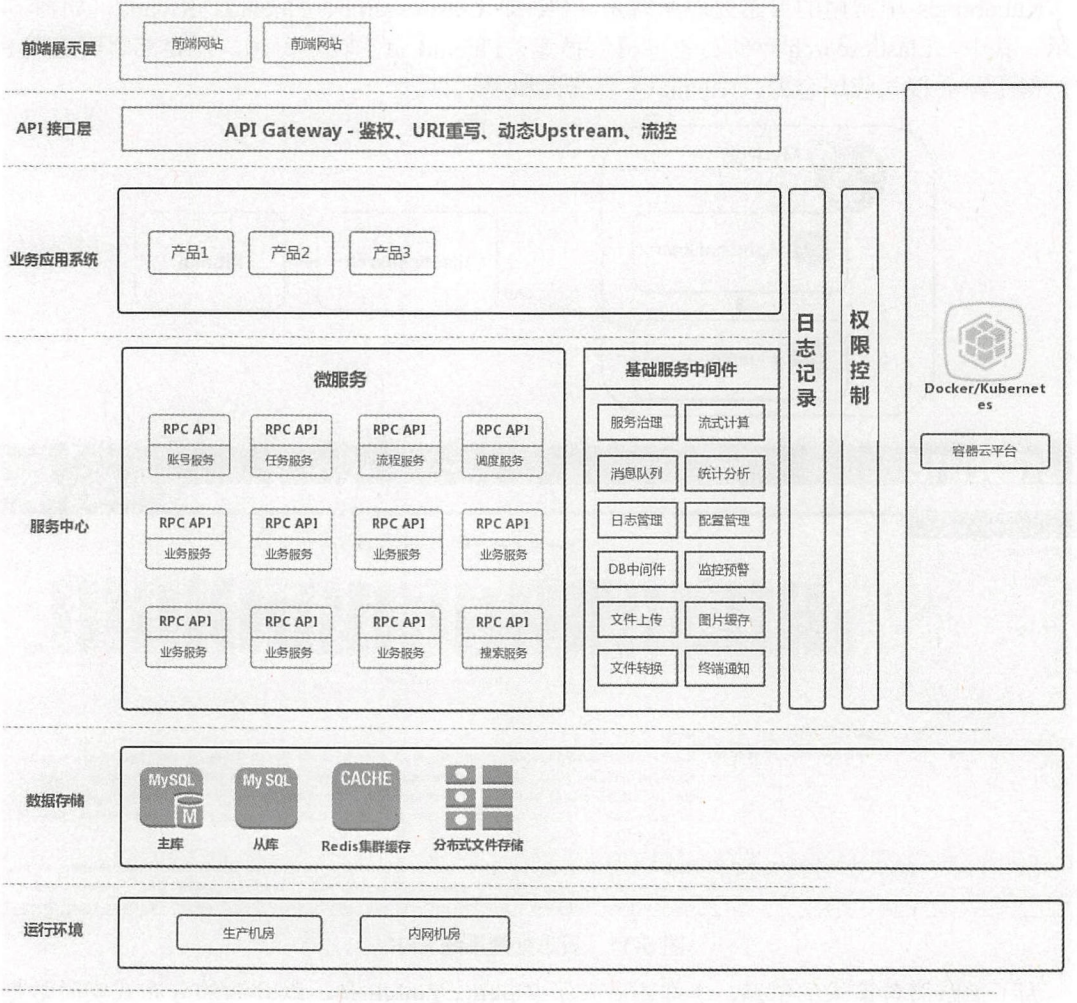


图 5-12　个推基于 Kubernetes 的微服务架构

5.4 总结

个推在微服务实践过程中，越来越觉得 Kubernetes 和 Docker 是非常适合微服务的，Docker 可以很好地进行环境的隔离并且粒度很小，部署到 Docker 容器中微服务之间可以很好地隔离，互不影响，同时服务的伸缩也可以很方便实现。而 Kubernetes 本身的设计思想就和微服务很契合，Servcie、Kube-DNS 实现了服务的注册和发现，HA 实现了服务的伸缩，Kubernetes 也有 ConfigMap 实现配置中心的功能。一方面 Kubernetes 提供的插件，很好地满足了服务监控、日志处理等需求；另一方面，Kubernetes 还提供了强大的功能进行容器的管理，使得容器调度、扩容缩容、故障恢复都能很方便地实现。随着微服务在个推的深入实践，还有很长的路要走，基于 Kubernetes 和 Docker 的 DevOps 平台的搭建，下一代的微服务框架 Istio 的尝试也都在规划之中。

本章作者：俞锋锋。

CHAPTER

6

第6章 使用Kubernetes进行交换机端口流量采集

本章主要围绕 Prometheus 在 Kubernetes 上的应用展开。首先对 Prometheus 进行简单介绍，并对其使用方法进行讲解；然后通过公司实际中的项目“流量采集系统”介绍，讲解如何在 Kubernetes 中将 Prometheus 应用到实际生产中。

6.1　Prometheus 简介与使用

Prometheus 是由 SoundCloud 开发的开源监控报警系统和时序列数据库（TSDB）。自 2012 年起，许多公司及组织已经采用 Prometheus，并且该项目有着非常活跃的开发者和用户社区。现在已经成为一个独立的开源项目，并且保持独立于任何公司。Prometheus 在 2016 年加入 CNCF，作为在 Kubernetes 之后的第二个由基金会主持的项目。

6.1.1　Prometheus 特点

Prometheus 主要有以下特点：

- 多维数据模型（时序列数据由 metric 名和一组 Key/Value 组成）。
- 在多维度上灵活的查询语言（PromQL）。
- 不依赖分布式存储，单主节点工作。
- 通过基于 HTTP 的 pull 方式采集时序数据。
- 可以通过中间网关进行时序列数据推送（pushing）。
- 目标服务器可以通过发现服务或者静态配置实现。
- 多种可视化和仪表盘支持。

6.1.2　Prometheus 相关组件

Prometheus 生态系统由多个组件组成，其中许多是可选的，这里针对一些重要的组件进行介绍。

- Prometheus 主服务：用来抓取和存储时序数据。
- Client Library：用来构造应用或 exporter 代码（Go、Java、Python、Ruby）。
- push 网关：用来支持短期的任务。
- Alertmanager：处理告警。
- 多种导出工具：可以支持一些特殊需求的数据出口（用于 HAProxy、StatsD、Graphite 等服务）。

6.1.3　Prometheus 架构

Prometheus 的各个组件基本是用 Golang 语言编写的，作为静态库编译和部署十分方便。Prometheus 整体架构如图 6-1 所示。

Prometheus 从被 Client Library 构造的任务获取 Metrics，可以直接使用 Pull 方式，或者对于短期的任务可以通过 Pushgateway。Prometheus 将抓取来的 sample 存储到本地，然后通过 rules 聚合及存储新的数据，或者产生告警。Grafana 或其他一些客户端可以通过调用 Prometheus API 图形化地显示这些数据。

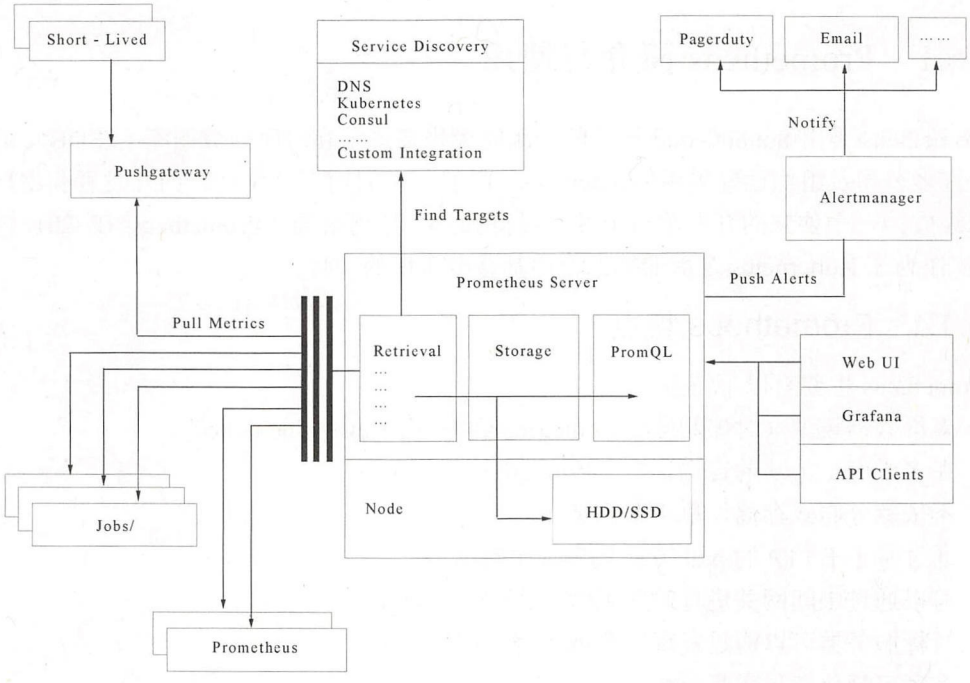


图 6-1　Prometheus 整体架构

6.1.4　Prometheus 适用场景

Prometheus 非常适合存储单纯的时序性数据。它既适合基于服务器监控，也适合动态的基于服务的架构。对于微服务来说，它支持收集各种数据，同时搜索也是强项。

Prometheus 在设计时考虑到可靠性，方便用户出错时可以马上解决问题。每个 Prometheus Server 都是独立的，不依赖网络存储或其他远程的一些服务。即使其他资源出问题了，也可以依靠它，而且也不需要其他资源就能使用。

Prometheus 的主要价值是可靠性，即使出现错误，也可以看到系统中的部分数据。如果用户需要百分之百的精确性，比如针对每个请求计费，Prometheus 是不够精准的。在这种情况下，可以使用其他系统手机和分析数据计费，Prometheus 用于监控。

6.1.5　Prometheus 的安装及使用

Prometheus 通过被抓取对象暴露出的 HTTP 端口抓取 Metrics。

1. Prometheus 的安装

首先下载最新版本（https://prometheus.io/download），并解压。

```
tar xvfz prometheus-2.2.1.linux-amd64.tar.gz
```

接着修改 Prometheus 默认的配置文件 prometheus.yaml。

```
# my global config
global:
  scrape_interval:     15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
```

```
    - static_configs:
      - targets:
        # - alertmanager:9093

  # Load rules once and periodically evaluate them according to the global 'evaluation_ interval'.
  rule_files:
    # - "first_rules.yml"
    # - "second_rules.yml"

  # A scrape configuration containing exactly one endpoint to scrape:
  # Here it's Prometheus itself.
  scrape_configs:
    # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
    - job_name: 'prometheus'

      # metrics_path defaults to '/metrics'
      # scheme defaults to 'http'.

      static_configs:
        targets: ['localhost:9090']
```

如上所示，其配置由 4 部分组成，分别是 global、alerting、rule_file、scrape_configs。

- global：是关于 Prometheus Server 的全局配置。这里有两个设置，分别是 scrape_interval 和 evaluation_interval。其中，scrape_interval 用于控制抓取 target 的频率。默认是 1 分钟，可以在每个 target 里面设置，覆盖当前的全局配置。当前设置为 15 秒。evaluation_interval 用来设置执行 rules 的频率，Prometheus 用 rules 创建新的时序数据和产生告警。
- alerting：用于配置告警相关，配置 Alertmanager 的地址。
- rule_files：用于指定 Prometheus 需要加载哪些文件，当前设置为空。只加载一次，执行计算的频率是由上面的 evaluation_interval 指定，每个 rule 里面也可以配置进行覆盖。
- scrape_configs：配置哪些资源需要被监控。因为 Prometheus 通过 HTTP，本身提供了自己的一些数据可以让 Prometheus 获取，在默认的配置中，有一个 job 叫作 Prometheus，抓取自己暴露的数据。这个 job 包含了一个静态的 target 是 localhost，端口是 9090。Prometheus 将自己的 Metrics 暴露在路径/metrics 中。因此这个默认的 job 数据抓取可以通过 url http://localhost:9090/metrics 获取。返回的数据主要用于当前 Prometheus 的状态和性能。具体可以参考配置文档。

2. 使用 Prometheus

1）Prometheus 监控 node

可以通过如下命令启动 Prometheus。

```
./prometheus --config.file=prometheus.yml
```

Prometheus 本身自带 exporter，所以可以直接通过 localhost:9090 监控 Prometheus 本身。同时可以使用 node exporter 监控任何 Linux 或 OS X 主机，如下所示。

首先需要在要监控的节点上安装 node exporter。

下载地址（https://prometheus.io/download#node_exporter）

```
tar zxvf node_exporter-0.16.0.linux-amd64.tar.gz
./node_exporter &
http://127.0.0.1:9100/metrics
```

配置 Prometheus 监控节点。

在 Prometheus.yaml 中的 scrape_configs 中添加如下。

```
- job_name: node
```

```
  static_configs:
    - targets: ['localhost:9100']
```

这个job名称为node，抓取静态的target，即localhost的9100端口数据，默认路径是metrics。一个重要的metric是up，表示target是否抓取成功，如果up是1，表示抓取成功，如果up是0，则表示失败。对于每个target都有一个up，当前配置一个up是Prometheus Server，另外一个是node exporter。

```
up{instance="localhost:9100",job="node"}
up{instance="localhost:9090",job="prometheus"}
```

Prometheus高可靠：同时将几台Prometheus Server运行在不同的机器中。相同的告警会被Alertmanager促发。对于Alertmanager的高可靠，可以运行多个实例，然后配置Prometheus Server给每个实例发送通知。

通过给Prometheus进程发送SIGHUP或http post请求/-/reload可以重新加载Prometheus的配置文件（--web.enable-lifecycle）。不同的组件会处理错误的更改。

通过node exporter监控主机级别Metrics，通过SNMP exporter监控支持SNMP的网络设备。

Prometheus存储所有的数据为时序，属于相同Metric和相同的label的不同时间的数据。除了存储时序，Prometheus可以根据查询结果产生临时的时序。每个时序由唯一的metri name和一系列Key-Value对，也叫label组成。metric name指定了主要的监测内容，比如http_request_total表示收到的所有的HTTP请求。label使得Prometheus支持多维度数据，相同的metri name和不同的label会产生一个特定的Metric实例。查询语句可以根据维度进行过滤或者聚合。修改、增加或删除任何的label都会创建新的时序。

每个暴露端口被抓取的点称之为instance，通常它对应一个进程。多个instance组合在一起（为了可靠性或者扩展性），称为job。如下所示。

```
job: api-server
instance 1: 1.2.3.4:5670
instance 2: 1.2.3.4:5671
instance 3: 5.6.7.8:5670
instance 4: 5.6.7.8:5671
```

当Prometheus从target获取Metrics时，它会自动地额外将label添加到该对象，分别是job和instance，job是该target属于的job；instance是target的URL部分<host>:<port>。如果其中任何一个已经存在了，就需要依赖honor_labels配置，具体可参考抓取配置文档。对于每个instance的抓取，Prometheus存储采样下面几个时序：

- up{job="<job-name>", instance="<instance-id>"}：可达为1，不可达为0。
- scrape_duration_seconds{job="<job-name>",instance="<instance-id>"}：抓取持续时间。
- scrape_samples_post_metric_relabeling{job="<job-name>",instance="<instance-id>"}：在metric relabel后剩余采样数量。
- scrape_samples_scraped{job="<job-name>",instance="<instance-id>"}：rarget暴露的采样数。

2）Prometheus配置文件

Prometheus的配置文件详解如下。

- --config.file：指定需要加载的配置文件。
- scrape config：scrape_config指定了一系列的对象及如何抓取它们的参数。通常一个scrape config指定了一个job，当然复杂情况下会不一样。当采集对象为固定的时，可以通过static_configs参数指定；当采集对象为动态的时，可以通过Prometheus支持的服务发现机制。

- relabel_configs：允许在任何一个 target 的 label 在采集前被修改。
- tls_config：可以支持 tls。
- azure_sd_config：允许抓取对象为 azure 的 vm。
- consul_sd_config：可以根据 consul's catalog api 获取采集 target。
- ec2_sd_config：可以将 AWS 上的 EC2 作为 target。
- openstack_sd_confg：可以将 OpenStack 上的 nova 作为 target。
- gce_sd_config：可以将 GCP 上的 GCE 作为 target。
- kubernetes_sd_config：可以通过 Kubernetes 的 RestAPI 接口获取集群各种状态，具体可以获取以下 Node 节点、Service 服务 Pod、endpoints、ingress。
- static_config：静态配置一系列的 targets 及它们共用的 label。在通常情况下，在抓取配置中指定静态对象。
- relabel_config：在对象被抓取之前，可以重新设置对象的 label，每个 scrape 的配置可以多次配置。根据配置文件中出现的先后顺序生效。除了每个 target 各自的 label，会有一个 label 为 job，值为当前 job_name，__address__ label 会设置成对象的<host>:<port>。relabel 之后，如果 instance 没有被设置，instance 这个 label 被设置成__address__，__scheme__和__metrics_path__设置为被采集对象的 scheme 和 metrics 的路径。__param_<name> label 会被设置成请求参数首次出现名为<name>，__meta__开始的 label 都可以在 relabel 时候被使用。以__开头的 label 在 relabel 完成后，都会从 label 集合中删除。如果在 relabel 时候需要临时用 label，则可以用__tmp 开始的 label，Prometheus 不会使用它们。
- source_labels：可以指定从当前 label 获取部分 label 的值，用 separator 进行分隔。
- target_label：将上面的值赋值给哪些 label，当替换时候必须指定。
- regex：正则表达式过滤 source_labels 的值。
- replacement：正则表达式匹配结果，默认为$1。
- action：基于正则结果需要的操作，默认 replace。可选值如下：
  - replace：在 source_labels 上正则结果，target_label 被设置为 replacement。
  - keep：丢弃正则不匹配的 source_labels。
  - drop：丢弃正则匹配的 source_labels。
  - labeldop：正则匹配 label 名称，如果匹配则移除。
  - labelkeep：正则匹配 label 名称，不匹配则移除。
- metric_relabel_configs：是在被存储前最后的一步，和上面 target relabel 语法类似。它无法作用在自动产生的时序，比如 up。一个用法是过滤不需要的时序。
- alert_relabel_configs：用于在发送给 Alertmanager 之前，和 target relabel 语法类似。alert relabel 作用在 external label 之后。一个例子是为了实现 HA，配置不同的 external label 发送相同的 alert。
- alertmanager_config：指定了发送 alert 出去的 Alertmanager 的地址，当然也允许配置参数和 Alertmanager 进行通信。Alertmanager 可以静态配置，也可以动态发现方式配置。
- remote_write：Prometheus 提供的功能，它支持把收集到的 Metrics 远程写入远端系统（长期存储系统）。
- write_relabel_configs：在发送给 remote endpoint 前 relabel，write relabel 也是在 external label 之后，可以用于限制发送哪个采样。

注意：其中 label keep 和 labeldrop 需要慎重使用，在移除后，Metrics 要唯一。

3）Prometheus rulers

Prometheus 支持两种类型的 rules，这两种配置后，会每隔一段时间执行。分别是 recording 和 alerting。在 Prometheus 中加载配置需要创建一个文件，写好内容后，通过 Prometheus 的 rule_files 加载。

该 rule 文件也可以发送 SIGHUP 给 Prometheus 进程。可以通过 promtool 来实现，如下：

```
go get github.com/prometheus/prometheus/cmd/promtool
promtool check rules /path/to/example.rules.yml.
```

Recording rules 可以提前计算那些常用的，或者计算起码比较耗时的，将它们作为新的时序。这样在查询时不需要重新计算，直接获取就可以。对于 dashboard 特别实用，每次更新页面，都会用同样的方式查询。

recording 和 alerting rules 存在于 rule group 中，在一个组里面的 rules 每隔一定时间顺序执行。

```
groups:
  - name: example
    rules:
    - record: job:http_inprogress_requests:sum
      expr: sum(http_inprogress_requests) by (job)
```

上面有一个 example group，在该文件中名称必须唯一，interval 默认为 global.evaluation_interval。rules 中有一个为 record，名称为 job:http_inprogress_requests:sum，它的值来源为 expr，产生新的 metric，在存储之前还可以添加或者覆盖 label。

alerting rules 定义条件，基于 expr 结果，发送告警给外部服务。

- alert：名称。
- expr：PromQL 表达式。
- for：持续多久促发告警。
- labels：添加或者覆盖 label。
- annotations：对于 alert 添加 annotations。

```
groups:
- name: example
  rules:
  - alert: HighErrorRate
    expr: job:request_latency_seconds:mean5m{job="myjob"} > 0.5
    for: 10m
    labels:
      severity: page
    annotations:
      summary: High request latency
```

for 会在每次执行 rule 的时候，在促发告警前检测告警是否持续 10 分钟。告警是活动的，但是还没触发，属于 pending 状态。

label 和 annotations 可以 templated。$labels 变量是 label 的 key/value，$value 保存值。

```
groups:
- name: example
  rules:

  # Alert for any instance that is unreachable for >5 minutes.
```

```
    - alert: InstanceDown
      expr: up == 0
      for: 5m
      labels:
        severity: page
      annotations:
        summary: "Instance {{ $labels.instance }} down"
        description: "{{ $labels.instance }} of job {{ $labels.job }} has been down for more than
5 minutes."

    # Alert for any instance that has a median request latency >1s.
    - alert: APIHighRequestLatency
      expr: api_http_request_latencies_second{quantile="0.5"} > 1
      for: 10m
      annotations:
        summary: "High request latency on {{ $labels.instance }}"
        description: "{{ $labels.instance }} has a median request latency above 1s (current value:
{{ $value }}s)"
```

Prometheus可以发现什么有问题了，但是还不能完全用于告警需求。需要另外一个模块负责总结，限制告警速率，禁止发送告警等各种需求。在Prometheus生态中，这就是Alertmanager。因此Prometheus只需要将alert发送到Alertmanager，Alertmanager会负责接下去的各种调度等。

4）Prometheus查询

Prometheus提供了查询语句，实时地查询或者聚合时序数据。可以将结果直观地显示或通过提供的HTTP API接口接入第三方的查询。

①表达式的查询结果一般是下面四种类型。

- instant vector：某个时刻，多个时序。
- range vector：多个时刻，多个时序。
- Scalar：一个浮点值。
- String：字符串。

instant vector选择在某个时刻筛选一系列的时序。最简单的是指定Metric名称。比如http_requests_total，也可以加{}进行额外的筛选，比如http_requests_total{job= "Prometheus", group="canary"}，其中 = 表示绝对相等，! = 表示不相等，=~ 表示正则约等，! ~ 表示正则不约等。http_requests_total{environment=~"staging|testing|development", method!="GET"}选择method不是GET，environment是staging、testing或development。

range vector选择某个范围内的采样数据。范围用[]表示，单位可以是s（秒）、m（分）、h（小时）、d（天）、w（周）、y（年）。http_requests_total{job="prometheus"}[5m]表示最近5分钟所有job是Prometheus的http_request_total的时序。可以用offset调整采样起始点。http_requests_total offset 5m表示前5分钟的http_requests_total的值，offset必须是紧跟着选择，例如sum(http_requests_total{method="GET"} offset 5m)是正确的，sum(http_requests_total{method="GET"}) offset 5m是错误的。rate(http_requests_total[5m] offset 1w)表示1个星期前的5分钟速率。

Prometheus支持许多运算和聚合操作。有数值计算+、-、*、/、%、^，双目数据运算可以用在scalar/scalar、 vector/scalar、vector/vector三对。

两个scalar计算比较好理解，就是普通两个值进行运算。

一个instant vector和一个scalar，如果乘以2，则每个原始vector里面的值都乘以2。

两个instant vector之间，左边的vector每一个值及右边每个匹配的值。重新组合产生新的vector，metric名称会被去掉。右边不匹配的不会出现在结果中。

②逻辑计算

➢ 双目比较运算符：==、!=、 >、 <、 >= 、<=。

➢ 逻辑双目运算符：and or unless。

vector1 and vector2：在 vector2 里面在的 vector1，其他被丢弃，名称还是左边的。

vector1 or vector2：左右加起来的 set。

vector1 unless vector2：左边元素在右边不存在的，存在的会被丢弃。

vector 匹配：one to one、many to one 或 one to many。

one to one：两边都有相同的 label 及值，ignoreing 可以忽略某些 label 和值，on 则是需要哪些 label 和值。

```
method_code:http_errors:rate5m{method="get", code="500"}  24
method_code:http_errors:rate5m{method="get", code="404"}  30
method_code:http_errors:rate5m{method="put", code="501"}  3
method_code:http_errors:rate5m{method="post", code="500"} 6
method_code:http_errors:rate5m{method="post", code="404"} 21

method:http_requests:rate5m{method="get"}  600
method:http_requests:rate5m{method="del"}  34
method:http_requests:rate5m{method="post"} 120

method_code:http_errors:rate5m{code="500"}/ignoring(code)method:http_requests:rate5m
{method="get"}  0.04            //  24 / 600
{method="post"} 0.05            //   6 / 120
```

many-to-one 和 one-to-many 匹配一边每个元素可以和另外一边多个元素匹配，可以使用 group_left 或 group_right，决定了哪边的级别高。

```
method_code:http_errors:rate5m/ignoring(code)group_left method:http_requests:rate5m
{method="get", code="500"}  0.04            //  24 / 600
{method="get", code="404"}  0.05            //  30 / 600
{method="post", code="500"} 0.05            //   6 / 120
{method="post", code="404"} 0.175           //  21 / 120
```

③聚合操作

Prometheus 支持下面内置的聚合操作：sum 多个合起来，不是值相加；min 最小、max 最大、avg 平均、count 计数。

假设 http_requests_total 有 label application、instance、group。可以计算每个 application 和 group 的和，sum(http_requests_total) without (instance)等价于 sum(http_requests_total) by (application, group)。计算所有的和，sum(http_requests_total)，获取最大的 5 个 topk(5, http_requests_total)。

Prometheus 支持一些函数。有些函数有默认的参数，比如 year(v=vector(time())instant-vector)，就是默认有 instanat vector，如果不提供，默认就是 vector(time())。abs 为绝对值，absent 为空，ceil 为取整，rate()为速率，avg_over_time(range-vector)为平均值等。

举例说明：

http_requests_total：返回所有 metric 名称为 http_requests_total 的时序。

http_requests_total{job="apiserver", handler="/api/comments"}：返回所有 metric 名称为 http_requests_total 及 label 含 job 是 apiserver，handler 为 api/comments 的时序。

http_requests_total{job="apiserver",handler="/api/comments"}[5m]：返回最近 5 分钟的，变成了 range vector，range vector 无法直接作图显示，但是可以在 tabular 显示。

http_requests_total{job=~".*server"}：模糊匹配所有 server 结尾的 job。

http_requests_total{status!~"4.."}：所有非 4xx 状态。

rate(http_requests_total[5m])：最近 5 分钟的速度。

sum(rate(http_requests_total[5m])) by (job)：所有实例最近 5 分钟的速度，保留 job 维度。

topk(3, sum(rate(instance_cpu_time_ns[5m])) by (app, proc))：获取排名前 3 的 CPU 使用率。

Prometheus 支持 HTTP 接口获取数据，具体看官方文档。

5）Prometheus 存储方案

Prometheus 包含一个存储在本地磁盘上的方案，同时也支持和其他存储系统对接。

本地存储文件：最终的数据包含很多的块，每 2 小时打包成一个块。每 2 小时的块包含一个目录，里面存了一个或者多个簇文件，那些文件是该时刻的时序数据，以及 metadata 文件和 index 文件（metric name 和名称关联到簇文件）。当前进来的数据保存在内存里，还没有被永久保存。不过也是足够安全的，它是基于 wal，即使 Prometheus 宕机后重启，也会根据 wal 文件重新正常工作。当时序被通过 API 删除后，删除的是记录在各自的 tombstone 文件里，而不是直接删除簇文件里面的内容。

```
./data/01BKGV7JBM69T2G1BGBGM6KB12
./data/01BKGV7JBM69T2G1BGBGM6KB12/meta.json
./data/01BKGV7JBM69T2G1BGBGM6KB12/wal
./data/01BKGV7JBM69T2G1BGBGM6KB12/wal/000002
./data/01BKGV7JBM69T2G1BGBGM6KB12/wal/000001
./data/01BKGTZQ1SYQJTR4PB43C8PD98
./data/01BKGTZQ1SYQJTR4PB43C8PD98/meta.json
./data/01BKGTZQ1SYQJTR4PB43C8PD98/index
./data/01BKGTZQ1SYQJTR4PB43C8PD98/chunks
./data/01BKGTZQ1SYQJTR4PB43C8PD98/chunks/000001
./data/01BKGTZQ1SYQJTR4PB43C8PD98/tombstones
./data/01BKGTZQ1HHWHV8FBJXW1Y3W0K
./data/01BKGTZQ1HHWHV8FBJXW1Y3W0K/meta.json
./data/01BKGTZQ1HHWHV8FBJXW1Y3W0K/wal
./data/01BKGTZQ1HHWHV8FBJXW1Y3W0K/wal/000001
./data/01BKGV7JC0RY8A6MACW02A2PJD
./data/01BKGV7JC0RY8A6MACW02A2PJD/meta.json
./data/01BKGV7JC0RY8A6MACW02A2PJD/index
./data/01BKGV7JC0RY8A6MACW02A2PJD/chunks
./data/01BKGV7JC0RY8A6MACW02A2PJD/chunks/000001
./data/01BKGV7JC0RY8A6MACW02A2PJD/tombstones
```

早期的 2 小时块的会被压缩到更长时间的块中。本地存储方式由于不是集群或者多备份，因此在节点异常或者磁盘出错时可能会丢失数据。如果可靠性要求不是很高，可以一直保存在本地。

--storage.tsdb.path：该参数可以指定存储的目录，默认是 data/。

--storage.tsdb.retention：指定多久移除旧数据。默认是 15 天。

平均下来，每个样本会占据 1～2 个字节，因此可以根据公式规划存储大小。

需要磁盘空间 = retention×每秒采样数×每个样大小。

如果想调优每秒采样速度，可以减少抓取量，减少 target 或者每个 target 的时序。或者提高抓取频率的值，间隔更久。当然，减少时序是更有效的方法。

当本地存储因为某些原因出问题时，应该马上停止 Prometheus，然后移除该目录。当然也可以通过移除部分块的目录解决问题，也就是会丢失 n×2 小时的数据。本地存储方式不建议用作长期存储。

第三方存储系统：考虑到本地存储的可扩展性和可靠性，Prometheus 没有开发长期存储系统，而是用了一系列接口和第三方存储集成。

Prometheus 通过两种方式与第三方存储系统集成：可以将采样通过一个远程 URL，通过某种格式写过去；可以通过一个远程 URL 读取采样。Prometheus 存储架构如图 6-2 所示。

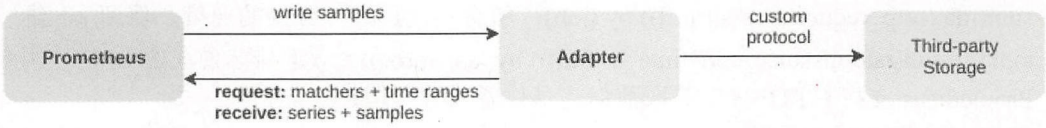


图 6-2　Prometheus 存储架构

3. Prometheus 注意事项和联邦应用

应避免很慢及很费力的查询，如果一个查询需要大量的数据，则对其画图就需要占用 Server 和浏览器资源。如果确实需要很多计算，则建议用 Recording rule。如果需要很多数据进行聚合，即使产生少量的新时序也会占据 Server 资源。

1）联邦：允许一个 Prometheus Server 抓取其他 Prometheus 的时序。可以适用从其他 Prometheus 抓取 metrics 的场景。

2）分层联邦：Prometheus 扩展到成百上千节点，类似一棵树，上层的 Server 向其子 Server 要数据。

3）跨 Service 联邦：一个 Prometheus 的某个服务被配置抓另外一个 Prometheus Server，只需要在这台新的上面同时促发告警及查询。

6.1.6　Prometheus SNMP Exporter

有时需要监控某些无法抓取的对象，可以使用 Prometheus pushgateway，允许用户将某些时序 push 到 pushgateway，然后用 Prometheus 抓取。有很多库或者 Server 可以产生 metrics 作为 Prometheus 的 metrics。有数据库、硬件相关、消息系统、存储、HTTP、APIS、日志、其他监控系统等各种类型。本节主要分析 SNMP Exporter。

SNMP Exporter 可以通过 SNMP 协议采集数据，然后给 Prometheus。主要由两部分组成，一个 Exporter 抓取，另外一个是产生配置给 Exporter。允许 SNMP Exporter 之后，可以通过 http://localhost:9116/snmp?target=1.2.3.4 方式获取 metric，其中 1.2.3.4 是 SNMP 设备的地址，也可以配置 module 参数，使用配置文件中的某个 module。SNMP Exporter 默认读取 snmp.yml 作为配置，这个配置不应该手动生成，而应通过 generator 产生配置文件。

```
scrape_configs:
  - job_name: 'snmp'
    static_configs:
      - targets:
        - 192.168.1.2  # SNMP device.
    metrics_path: /snmp
    params:
      module: [if_mib]
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: 127.0.0.1:9116  # The SNMP exporter's real hostname:port.
```

上面这个配置首先配置了静态的 SNMP 设备为 192.168.1.2，然后配置了参数，使用 module 为 if_mib。每个 target 默认会有 instance 为__address__，这时如果不设置__address__，那么会被配置成每个 target 的地址，然而无法直接和 target 通信，因此将__address__设置为了 Exporter 的地址。如果这时不显示地设置 instance，那么所有的 instance 都会被设置为__address__地址，因此首先将采集 target 的地址即__address__赋值给__param_target，说明请求的时候带上参数为 target、值为 target 的值；同时 instance 设置为参数 target 的值。

6.1.7　Prometheus 告警

Prometheus 告警由两部分组成，分别是 Prometheus 中告警规则给 Alertmanager 发送告警。Alertmanager 管理这些告警，可能忽略、聚合、通过邮件或其他发送通知。主要流程是：设置并配置好 Alertmanager，配置 Prometheus 和 Alertmanager 通信，创建告警规则。

Alertmanager 处理来自 Prometheus 发送的告警，然后进行调度，分类然后根据配置发送给合适的接收者，比如邮件、pagerduty 或 opsgenie 等。当然也允许忽略告警。

1. 分组

很多相同类型的告警统一成单一的通知。通常用于同时发送很多故障，成百上千条告警会产生。比如一个服务的几百个实例运行在集群中，如果因为网络原因，有一半出了问题，如果每个都发送消息无法连接数据库，会有几百个告警发送给 Alertmanager。

作为一个用户，只需要一个简单页面就可以知道哪些服务实例受影响了。因此，Alertmanager 需要被配置成可以给告警分组，根据它们的集群及告警名称，从而发送一个通知。

对告警分类、分类告警的时间设置及接受告警，都是根据配置文件中的一颗路由树决定的。

2. 抑制告警

因为有些其他告警已经触发了，某些告警就不发送通知了。比如一个告警触发告诉整个集群不可达了。Alertmanager 可以配置成忽略其他所有的告警，如果告警被触发了。可以避免很多和真正原因无关的通知。

3. 禁止告警

某段时间内禁止告警。进来的告警匹配规则中设置的禁止，如果匹配了，就不会再发送告警对应的通知。

Alertmanager 可以通过命令行或者配置文件进行配置，命令行配置的是不可修改的系统参数，配置文件定义的是抑制规则，通知路由和通知接收者。Alertmanager 可以在运行时重新加载配置。如果配置有问题，则修改的不会被加载，会记录错误。可以通过给进程发送 SIGHUP 或/-/reload 重新加载配置。

4. 配置文件

通过使用参数—config.file 指定配置文件。

```
global:
  # ResolveTimeout is the time after which an alert is declared resolved
  # if it has not been updated.
  [ resolve_timeout: <duration> | default = 5m ]

  # The default SMTP From header field.
  [ smtp_from: <tmpl_string> ]
  # The default SMTP smarthost used for sending emails, including port number.
  # Port number usually is 25, or 587 for SMTP over TLS (sometimes referred to as STARTTLS).
  # Example: smtp.example.org:587
  [ smtp_smarthost: <string> ]
  # The default hostname to identify to the SMTP server.
  [ smtp_hello: <string> | default = "localhost" ]
  [ smtp_auth_username: <string> ]
  # SMTP Auth using LOGIN and PLAIN.
  [ smtp_auth_password: <secret> ]
  # SMTP Auth using PLAIN.
  [ smtp_auth_identity: <string> ]
  # SMTP Auth using CRAM-MD5.
  [ smtp_auth_secret: <secret> ]
```

```
    # The default SMTP TLS requirement.
    [ smtp_require_tls: <bool> | default = true ]

    # The API URL to use for Slack notifications.
    [ slack_api_url: <string> ]
    [ victorops_api_key: <string> ]
    [ victorops_api_url: <string> | default = "https://alert.victorops.com/integrations/ generic/
20131114/alert/" ]
    [ pagerduty_url: <string> | default = "https://events.pagerduty.com/v2/enqueue" ]
    [ opsgenie_api_key: <string> ]
    [ opsgenie_api_url: <string> | default = "https://api.opsgenie.com/" ]
    [ hipchat_api_url: <string> | default = "https://api.hipchat.com/" ]
    [ hipchat_auth_token: <secret> ]
    [ wechat_api_url: <string> | default = "https://qyapi.weixin.qq.com/cgi-bin/" ]
    [ wechat_api_secret: <secret> ]
    [ wechat_api_corp_id: <string> ]

    # The default HTTP client configuration
    [ http_config: <http_config> ]

  # Files from which custom notification template definitions are read.
  # The last component may use a wildcard matcher, e.g. 'templates/*.tmpl'.
  templates:
    [ - <filepath> ... ]

  # The root node of the routing tree.
  route: <route>

  # A list of notification receivers.
  receivers:
    - <receiver> ...

  # A list of inhibition rules.
  inhibit_rules:
    [ - <inhibit_rule> ... ]
```

route 模块定义了一个 Node 及其子节点在路由树中的信息。如果未指定可选参数，则继承于它的父节点。每个告警都进入路由树中最高级的路由，它和所有告警都匹配，接下来继续走子几点，如果 continue 设置成 false，那么找到一个就停止了，如果为 true，则继续找接下来的相邻节点。如果都没有找到，则告警会根据当前的 Node 配置信息进行处理。

```
  route:
    receiver: 'default-receiver'
    group_wait: 30s
    group_interval: 5m
    repeat_interval: 4h
    group_by: [cluster, alertname]
```

所有不符合下面子路由的，都会停留在当前节点，然后被调度到 default-reveiver。

```
  routes:
```

所有 service=mysql 或 service=Cassandra 的都被调度到 database-pager。

```
  - receiver: 'database-pager'
    group_wait: 10s
    match_re:
      service: mysql|Cassandra
```

所有是 team=frontend 的，加入 product 和 environment，而不是 cluster 及告警名称。

```
  - receiver: 'frontend-pager'
    group_by: [product, environment]
    match:
      team: frontend
```

receiver 集合一系列的通知，当前没有增加新的 receiver，可以通过 webhook 添加个性化

的通知。

6.1.8　Grafana

Grafana 支持多种不同的时序数据库数据源，Grafana 对每种数据源提供不同的查询方法，而且能很好地支持每种数据源的特性。当前支持 Graphite、InfluxDB、OpenTSDB、Prometheus、Elasticsearch 和 CloudWatch。当然，也可以自己开发对接特殊的数据。这里简单配置 Prometheus，如果 Prometheus 地址不属于浏览器直接能访问的网络，则可以配置 Proxy 模式，用 Grafana 访问，如图 6-3 所示。具体细节可以参考 Grafana 官方文档，这里主要根据当前项目需求来图形化显示。

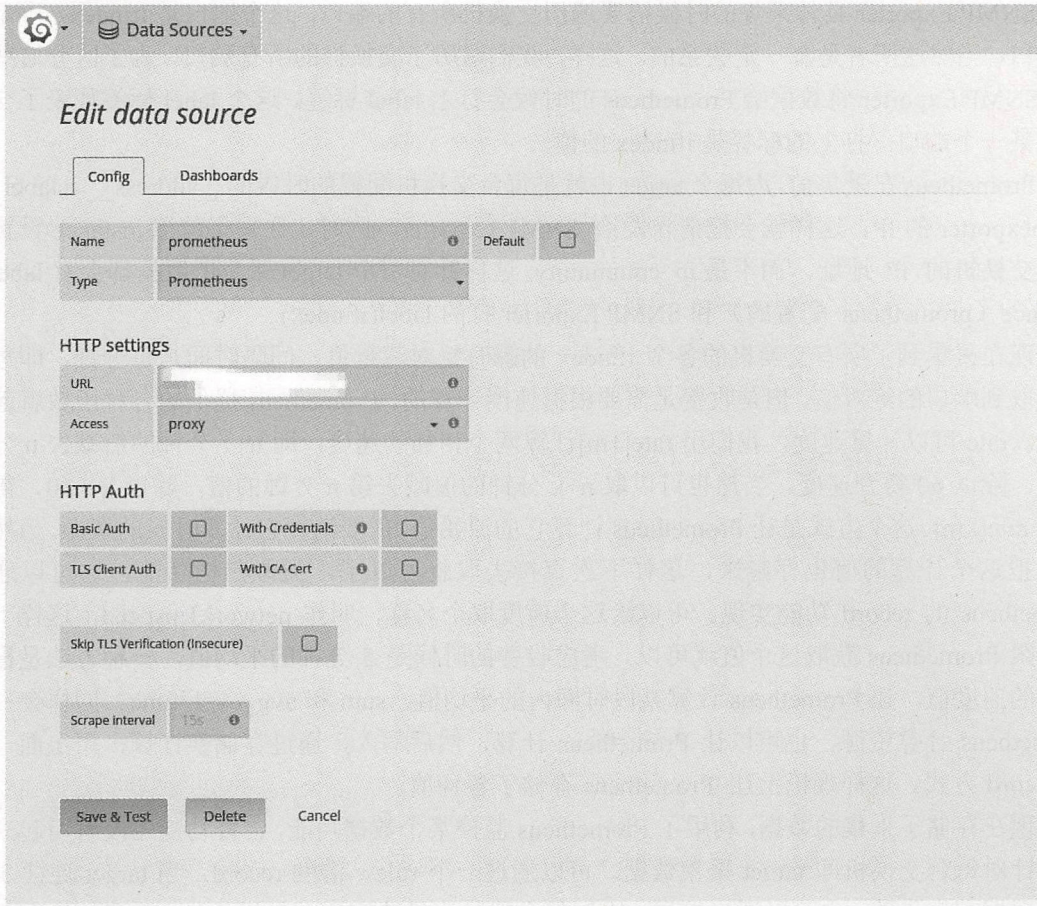


图 6-3　Grafana 数据源配置

6.2　流量采集系统

交换机端口流量采集，需求是统计每个端口的速度。当然，速度的精准度和向交换机要数据的频率相关，即 V(a)-V(b)/(a-b)，a 时刻减去 b 时刻的值，然后除以时间，就是速度。当 a 和 b 无限接近时，在一段时间内可以得到任何一个时刻的速度。在现实中不需要记录每秒的速度。因此，a 和 b 的间隔时间一般是 1 分钟、2 分钟、5 分钟。在实现方面，Zabbix 可以简单用来进行监控；复杂一点的需要对 Zabbix 进行二次开发，当然扩展不是很好。我们采用了

Prometheus+SNMP Exporter 实现。主要原理如下，Prometheus 根据配置文件定时地和 SNMP Exporter 通信，让它采集数据，然后 SNMP Exporter 将采集数据给 Prometheus，Prometheus 再把数据存储起来，如果 Prometheus 没有通知 SNMP Exporter 采集，SNMP Exporter 不会主动采集。在这里，我们使用了 Prometheus 的存储数据的力。

当 SNMP Exporter 采集数据时，官方的 SNMP Exporter 采集参数是采集对象，即交换机的 IP 地址，Prometheus 配置的 target 是一系列的 IP 地址。但实际情况是每台交换机的 community 会不一样。有两种解决办法，一个是所有交换机 community 设置一样，另一种是对 SNMP Exporter 进行二次开发。从合理性上来说，肯定需要二次开发，因此把原先的参数 IP 地址改成 IP 地址和 community 一起，SNMP Exporter 收到请求后，解析出 IP 和 comminity 进行采集。

SNMP Exporter 具体采集的时候需要遍历交换机所有的端口，这个端口是 ifindex，不是物理端口，当然这两者是有一定关系的。这样，我们遍历了交换机的所有端口，为了区分这些端口，SNMP Exporter 将数据给 Prometheus 的时候会打上 label 标签。这个 label 标签代表了交换机的某一个端口，打上的标签是 ifindex 的值。

Prometheus 在采集前，为每个 target 也就是每台交换机配置的时候的__address__ relabel 是 snmpexporter 的 IP，这样就会把请求发给 SNMP Exporter。同时，在采集前把 instance 设置为对应交换机的 IP 地址，而不是 ip_community。这样针对每个 target 采集的数据都会有 label，instance（prometheus 配置的）和 SNMP Exporter 打的 label(ifindex)。

现在采集到了所有交换机的各个 ifindex 的瞬时绝对流量值，间隔时间是每分钟，即每分钟获取到端口的绝对值。但是收费是需要根据速度计算的，Prometheus 提供的各种函数就起作用了。rate 可以计算速度，我们用 rate[1m]计算两个相邻的速度，即 n+1 分钟的值减去 n 分钟的值，除以 60 得到速度。当然也可以取 n+k 分钟的值减去第 n 分钟的值，除以 k×60，得到速度 rate[km]。这个方式是让 Prometheus 计算它的原始值，需要消耗它计算时候的资源。当然，可以把这个计算的值也存起来，这样下次直接去取就可以了，类似归档功能，这可以通过 Prometheus 的 record 功能实现。可以给这个速度取个名称，叫作 network1mspeed，这样下次直接跟 Prometheus 获取这个值就可以。考虑收费的时候是 5 分钟的平均值，一种方法是根据之前的速度值，让 Prometheus 计算某段时间内的平均值、sum 和 avg_over_time。同样会占用 Prometheus 计算资源，也可以让 Prometheus 计算，然后写入，通过存储换计算，用上面一样的 record 方式。这样理论上让 Prometheus 存储了各种值。

现在存储了采集的数据，利用了 Prometheus 监控某个数据功能，告警功能还没有用起来。上面针对每台交换机即 target 采集数据，可以通过一个 rule，借助 record，当 target 连续 3 分钟没数据产生告警。也可以适当扩展，比如某个端口流量速度达到怎样的时候告警，这会将业务和采集数据耦合。

现在有了所有数据，如何与公司内部 OA 系统对接呢？采集的端口并非是物理端口，但客户绑定的是交换机的物理端。也就是说交换机的物理端口和采集时候的端口有个转换过程。那么问题来了，端口关系从哪里来的呢？一种是看物理端口，比如 1 对应的虚拟端口是 54；物理端口 2 对应的是 55，依次类推，某个虚拟端口是 53。这是根据表象分析，理论上是不可取的。因此，分析最后找到了交换机物理端口和采集逻辑口内部所关联的，也就是通过 SNMP 接口找到它的采集 ifindex 口和物理端口关系，然后对其进行采集。

运维人员如何查看数据，也就是如何展示呢？借助 Grafana 把 datasource 配置到 Prometheus，

也就是 Grafana 向 Prometheus 获取数据。由于使用人员关注的是某些客户使用的端口，或者是一些重要交换机的重要端口。因此，Grafana 需要二次开发，支持根据选择条件动态地取数据并展示，如图 6-4 所示。

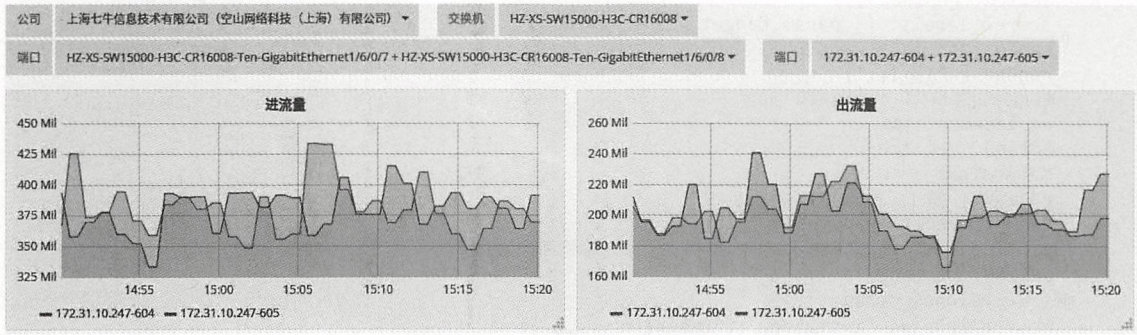


图 6-4　流量采集效果图

上面所说的那些客户是哪里来的呢？客户用了什么交换机的什么端口呢？这需要公司 OA 提供，公司 OA 在里面添加交换机，输入 IP 和 community 和 SNMP Exporter 通信获取到该交换机的所有物理端口及各种 ifindex，OA 里面端口进行关联客户，这样 OA 保存了客户在某段时间内用了某台交换机的某个端口。OA 服务提供 Grafana 当前端口在使用的客户端口情况。

```
global:
  scrape_interval: 1m
  scrape_timeout: 50s
  evaluation_interval: 1m
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - flow-2-netbank-flow-alertmanager
    scheme: http
    timeout: 10s
rule_files:
- /etc/config/alertmanager.rules.yaml
scrape_configs:
- job_name: prometheus
  scrape_interval: 1m
  scrape_timeout: 50s
  metrics_path: /metrics
  scheme: http
  static_configs:
  - targets:
    - localhost:9090
- job_name: ningbo
  params:
    module:
    - in_out
  scrape_interval: 1m
  scrape_timeout: 50s
  metrics_path: /snmp
  scheme: http
  static_configs:
  - targets:
- 192.168.2.123_helloworld
- 192.168.2.124_helloworld
- 192.168.2.125_helloworld
…
  relabel_configs:
```

```
    - source_labels: [__address__]
      separator: ;
      regex: (.*)
      target_label: __param_target
      replacement: $1
      action: replace
    - source_labels: [__param_target]
      separator: ;
      regex: (.*)[_](.*)
      target_label: instance
      replacement: $1
      action: replace
    - separator: ;
      regex: (.*)
      target_label: __address__
      replacement: 192.168.2.1:9116(exporter 地址和端口)
      action: replace
    metric_relabel_configs:
    - source_labels: [instance,  ifIndex]
      separator: ;
      regex: (.*)[;](.*)
      target_label: switchPort
      replacement: $1-$2
      action: replace
```

参考网站：

https://prometheus.io/docs/introduction/overview/。

https://www.opsdev.cn/post/Prometheus-first-exploration.html?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io。

本章作者：何晓波。

CHAPTER

7

第7章 搜道微服务容器化实践

聚客通是搜道网络技术有限公司（以下简称“搜道”）基于 SaaS 模式的新零售营销平台。随着平台的不断迭代，系统架构复杂度不断提高，人工维护部署成本日益增长。而平台面对的是商家服务，高峰期与低谷期之间的负载指数差值越来越大，要求有良好的弹性扩展、快速部署能力。

Docker 容器云平台是搜道运维技术团队为内部服务整合、开发的一套的容器管理平台，支持基础设施私有云和公有云对接，实现云上和云下实例使用一套平台进行管理，业务实例按需弹性扩容和缩容，规范化的项目管理流程、测试、上线流程，旨在将开发人员、测试人员从基础环境的配置与管理中解放出来，使其更聚焦于自己的业务开发。

7.1 为何选择 Docker

7.1.1 公司架构演变过程

公司开创初始，业务较为单一，应用流量较低，单服务器就能够支撑起所有的功能，包含 Web 应用、数据库存储等。随着业务的发展，Web 应用的承载量达到单服务器的临界限，这时需要使用负载均衡将用户的请求均匀分散，使用 Nginx 作为代理服务器，通过 LBS 实现 Web 应用端的可扩展。但是，在业务不断变化的过程中，平台的功能越来越繁杂，核心功能与次要功能全部耦合在一起，造成了资源的浪费。参考目前合理的架构，采用了微服务架构作为底层服务支撑，以各功能模块为粒度划分，部署在不同的服务器上。当服务越来越多时，集群如何管理、如何高效地扩展成为核心问题，而随着.NETCore 的发布，跨平台部署成为可能，最终采用 Docker 作为容器部署。

7.1.2 平台存在的问题

1. 硬件资源利用率不够，造成资源的浪费

在网站功能中，不同的业务场景有计算型、I/O 读写型、网络型、内存型，集中部署应用就会导致资源利用率不合理。比如，一个机器上部署的服务都是内存密集型，CPU 资源很容易浪费。

2. 环境、版本管理复杂，缺乏上线部署流程，增加问题排查的复杂度

由于内部开发流程的不规范，代码在测试或者上线过程中，对一些配置项和系统参数进行随意的调整，在发布时进行增量发布，一旦出现问题，就会导致测试的代码和线上运行的代码不一致，增加了服务上线的风险，也增加了线上服务故障排查的难度。

3. 扩展能力较差，混合云模式难以快速复制

在开发过程中存在多个项目并行开发和服务的依赖问题，由于环境和版本的复杂性很高，不能快速搭建和迁移一个环境，导致在测试环境中无法模拟线上的流程进行测试。很多工程师在线上环境进行测试，有很高的潜在风险，同时导致开发效率降低。

4. 传统物理机启动过程较慢，管理复杂

传统虚拟机和物理机在启动过程进行加载内核，执行内核和 init 进程，导致在启动过程中占用时间较长，而且在管理过程中会遇到各种各样的管理问题。

7.1.3 容器优势

1. 易扩展

面向庞大的商家用户，在商家发布活动的时候，系统访问流量呈指数级增长，快速扩展是其中最大的一个因素。

2. 宿主依赖

平台基础架构依赖于 RPC 服务，每个模块为独立的微服务，各模块之间由于业务需要而互相依赖，容器化可以使得宿主机不需要关心某一个容器运行所需要的依赖。

3. 一致构建

基于 Dockerfile，实现可重复的自动化构建。

7.2 Docker 容器云架构方案

基于 Docker 容器技术，公司运维技术团队采用容器云平台，其整体架构如下：

1）基础设施。包含网络、服务器、存储等计算资源。

2）快速部署和高可用。底层采用 KVM，将 Docker 架设在 KVM 上，避免部署很多复杂的组件，结合 Ceph 分布式存储提供高可用环境。

3）弹性调度。Docker 容器云平台集群节点管理，镜像中心管理业务镜像，统一监控，统一日志管理，定时任务管理。

4）服务编排。服务注册、服务发现，容器节点在线的扩容和缩容，服务优雅上线，回滚降级，规范.NET、Node、Python 等规范化的上线。

5）统一门户。规范化整个业务流程，简洁的用户流程，可动态地管理整个云环境的所有资源。

7.2.1 技术选型及实践

1. 高可用

对于单点问题一直是一个难点，搜道采用 KVM+Ceph 分布式存储服务构建高可用、高性能的云平台。Ceph 是开源的分布式存储解决方案，支持对象存储、块存储和文件存储访问类型。

由于该存储主要用于系统运行与应用存储，磁盘压力相对来说不会很大，公司使用三个节点硬盘，采用固态硬盘+机械硬盘各两块混合模式，网络两块网卡组建 bond，如图 7-1 所示。建议根据自己的业务类型选择合适的硬件。

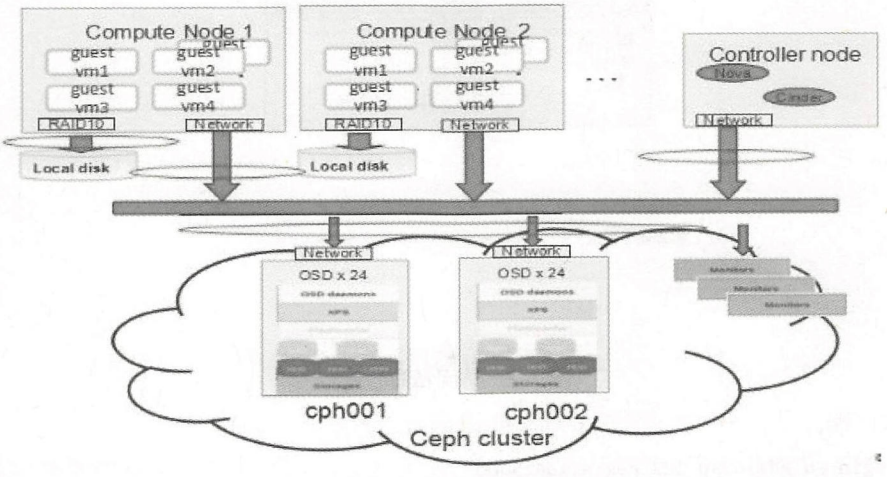


图 7-1 Ceph 架构图

2. 镜像标准

1）Docker Registry 介绍

Docker Registry 由三个部分组成：Index、Registry、Registry Client。

可以把 Index 认为是负责登录、认证、存储镜像信息和对外显示的外部实现，而 Registry 负责存储镜像的内部实现，Registry Client 则是 Docker 客户端。

2）私有仓库搭建

安装 Docker Registry，Docker 版本需要 1.6 以上。

```
docker pull registry
```

创建存放密码账号的文件，自定义账号密码。

```
mkdir -p /docker-hub/auth
docker run --entrypoint htpasswd registry -Bbn admin 123 > auth/htpasswd
```

启动 docker-hub 容器。

```
docker run -d -p 5000:5000 --restart=always --name docker-hub -v /docker-hub/registry:/var/lib/registry -v /docker-hub/auth:/auth -e "REGISTRY_AUTH=htpasswd" -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd registry
```

客户端没有登录的情况，pull、push 会提示报错，无法提交，需要登录私有仓库。

登录如下所示。

```
docker login -u admin -p 123 reg.a.lan:5000
```

退出如下所示。

```
docker logout reg.a.lan:5000
```

认证后无法直接在服务器查看 curl 127.0.0.1:5000/v2/_catalog 仓库的镜像，会报错，但是可以用浏览器访问，如图 7-2 所示。

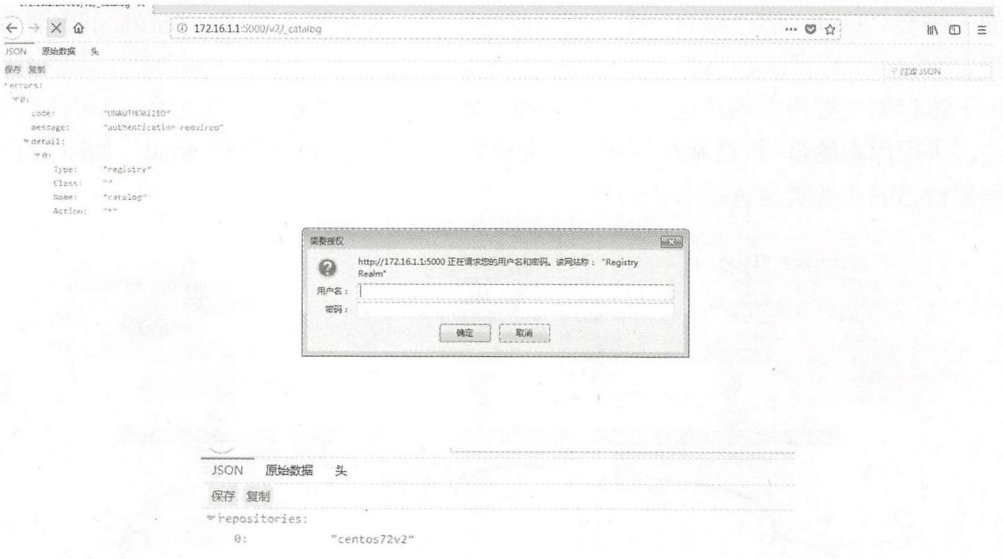


图 7-2　访问私有仓库

配置客户端。

```
docker login -u admin -p 123 reg.a.lan:5000
```

通过外网拉取一个镜像做测试。

```
docker pull busybox
```

打标签。

```
docker tag busybox reg.a.lan:5000/busybox
```

推送到仓库。

```
docker push reg.a.lan:5000/busybox
```

查看仓库，有了刚才推送的镜像。

```
http://reg.a.lan:5000/v2/_catalog
{"repositories":["busybox"]}
```

众所周知，Docker 的镜像是分层的。需要对镜像分层进行约定：第一层是操作系统层，由 CentOS 基础镜像构成，安装一些通用的基础组件；第二层是中间件层，根据不同的应用程序，安装它们运行时需要使用到的各种中间件和依赖软件包，如 Nginx、Tomcat 等；第三层是应用层，仅包含已经打好包的各应用程序代码。

做个简单分层例子：比如在镜像仓库中有一个 CentOS72v2 镜像为基础镜像，简单运行一个容器，将代码 hello 的代码文件载入镜像中，Docker 客户端通过镜像获取 hello 代码文件。具体操作如下：

获取基础镜像 CentOS72v2，直接运行容器，默认也会下载镜像。

```
[root@Docker ~]# docker run -itd --name=test01 --net=none reg.a.lan:5000/centos72v2 /bin/bash
```

将本地代码发布到新创建的容器中。

```
[root@Docker ~]# docker cp hello test01:/opt/
```

假如这个镜像代码测试没问题，就可以生成最新的应用镜像，推送到镜像仓库，利用 commit 扩展一个镜像，并推送到仓库。

```
[root@Docker ~]# docker commit -m "add hello" -a "ldt" 8b2aaa69e4bb reg.a.lan:5000/centos72v2:
1.0.1
sha256:4e07e163fbfea869e49c53d4d3cf0c4ac3b5d4376d565c299f4113caff2ff4c3
[root@Docker ~]# docker push reg.a.lan:5000/centos72v2:1.0.1
```

查看镜像仓库，然后运行新镜像，会自动地从私有镜像库中拉取。

```
[root@Docker ~]# curl http://reg.a.lan:5000/v2/centos72v2/tags/list
[root@Docker ~]# docker run -itd reg.a.lan:5000/centos72v2:1.0.1 /bin/bash
Unable to find image 'reg.a.lan:5000/centos72v2:1.0.1' locally
Trying to pull repository reg.a.lan:5000/centos72v2 ...
1.0.1: Pulling from reg.a.lan:5000/centos72v2
0519e7752724: Already exists
9dd97824f54c: Already exists
a78cafa70d59: Pull complete
Digest: sha256:9badc93a637b69c297af0e1c56fe8f1dbf34e852afc9479fe0d4e58f86b9c032
```

3. 编排工具（Rancher）

目前编排工具有 Swarm、Mesos、Kubernetes、Rancher 等，这里采用 Rancher。Rancher 具有图形化管理界面、部署简单、方便等特点，可以与 AD、LDAP、GitHub 集成，基于用户或用户组进行访问控制，快速地将系统的编排工具升级至 Kubernetes 或者 Swarm，同时由专业的技术团队进行支持，降低了容器技术入门的难度。

应用的容器实例进行统一的编排调度时，配合 Docker-Compose 组件，可以在同一时间对多台宿主机执行调度操作。同时，在服务访问出现峰值和峰谷时，利用特有的 rancher-compose.yml 文件调用“SCALE”特性，对应用集群执行动态扩容和缩容，让应用按需求处理不同的请求，如图 7-3 所示。

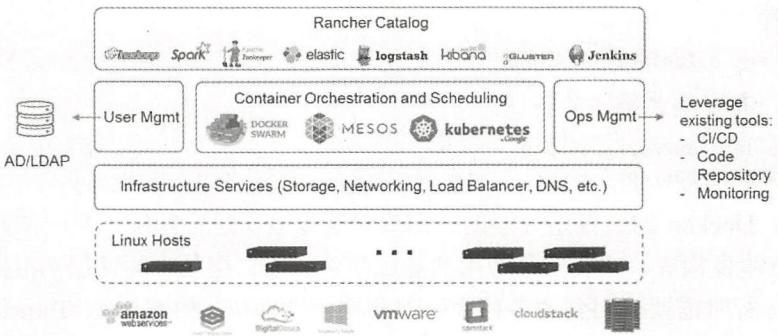


图 7-3　Rancher 主要组件与功能

4. 持续集成

在持续集成方面，公司还没有做到自动化，目前靠手动+工具实现，定义了持续部署规范的流程：

1）开发人员向 GitLab 拉取项目代码和配置项文件，执行编译任务。

2）将编译好的应用打包，通过运维团队开发的系统发布到定义好的目录。

3）拉取基础镜像。

4）根据当前应用及所属环境定制化生成 docker-compose.yml 文件，基于这个文件执行 rancher-compose 命令，将应用镜像部署到预发环境（发布生产前的测试环境，相关配置、服务依赖关系和生产环境一致）。

5）预发环境测试通过后，将应用镜像部署至线上环境，测试结果通知后端测试人员。

5. 监控管理

采用 Python，通过 Zabbix 自动发现实现监控 Docker 状态，通过脚本自动发现主机上的 Docker 容器，通过 Python 获取每个容器的系统状态，包括 CPU 使用率、内存使用率及网络资源使用率，如图 7-4、图 7-5 所示。

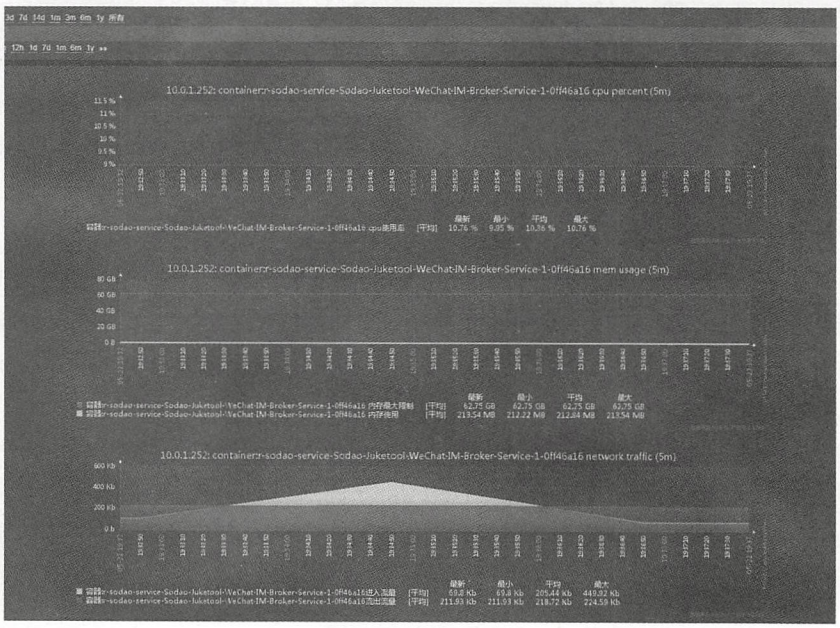


图 7-4　容器监控

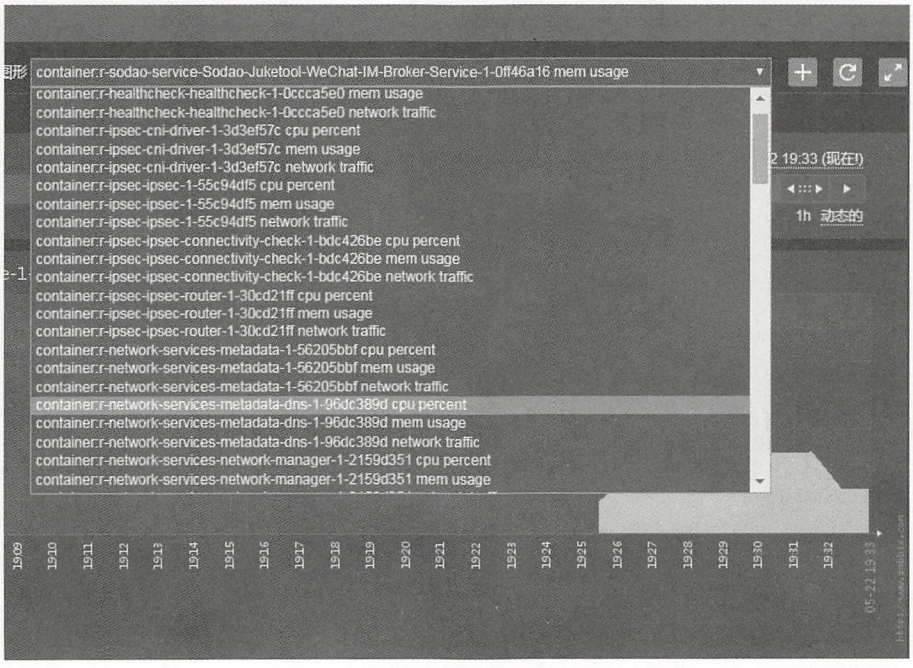


图 7-5　监控列表

6. 日志管理

容器在运行时会在只读层之上创建读写层，所有对应用程序的写操作都在该层进行。当容器重启后，读写层中的数据（包含日志）也会被一并清除。虽然可以通过将容器中的日志目录挂载到宿主机解决此类问题，但当容器在多个宿主机间频繁漂移时，每个宿主机上都会有留存应用名的部分日志，增加了开发人员查看、排查问题的难度。

综上所述，需要将日志统一管理。公司引入了 ELK 系统，将应用运行过程中产生的日志统一存储，并且支持多种方式的查询操作。

我们采用的方式是将日志统一写入数据库中，通过 Logstash 从数据库取出数据写入 Elasticsearch，通过 Kibana 查看日志。当然，写入时间的间隔是 1 分钟，如果需要实时查看，直接查询数据库即可。

具体代码如下所示。

```
input {
    stdin {
    }
    jdbc {
      # mysql jdbc connection string to our backup databse
      jdbc_connection_string => "jdbc:sqlserver://10.0.1.1:1433;databaseName=SD_SiteLog"
      # the user we wish to excute our statement as
      jdbc_user => "user"
      jdbc_password => "123456"
      # the path to our downloaded jdbc driver
      jdbc_driver_library => "/elk/service/logstash-2.3.3/sqljdbc4.jar"
      # the name of the driver class for mysql
      jdbc_driver_class => "com.microsoft.sqlserver.jdbc.SQLServerDriver"
      jdbc_paging_enabled => "true"
      jdbc_page_size => "50000"
      record_last_run => "true"
      statement_filepath => "/elk/service/logstash-2.3.3/conf/sqlserver.sql"
      use_column_value => "true"
```

```
        tracking_column => "id"
        last_run_metadata_path => "/tmp/myid_auto"
        schedule => "* * * * *"
        type => "windowslog"
    }
}

filter {
    json {
        source => "message"
        remove_field => ["message"]
    }
}

output {
    elasticsearch {
        hosts => "10.0.1.2:9200"
        index => "logstash-%{type}-%{+YYYY.MM}"
        document_id => "%{id}"
    }
}
```

sqlserver.sql 文件如下所示。

```
select * from person where person.id > :sql_last_value
```

7. 实践案例

1）固定软件版本信息。

- OS：采用 CentOS Linux release 7.4.1708（Core）系统。
- Docker：docker-ce-17.03.2.ce。
- Rancher：Rancher v1.6.14。

2）安装部署，推荐离线安装，版本一致避免未知问题。采用最小化安装即可。安装 Docker：

```
docker-ce-17.03.2.ce-1.el7.centos.x86_64.rpm
docker-ce-selinux-17.03.2.ce-1.el7.centos.noarch.rpm
```

下载地址为 https://code.aliyun.com/duantianliao/docker/blob/master/或 https://pan.baidu.com/s/1AkcYR20D1VqHGsnO2IJfdg。

```
[root@Docker02 ~]# yum -y localinstall docker-*
```

启动服务并查看 Docker 信息。

```
[root@Docker02 ~]# systemctl  start docker
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 17.03.2-ce
Storage Driver: overlay
 Backing Filesystem: xfs
 Supports d_type: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
```

```
Init Binary: docker-init
containerd version: 4ab9917febca54791c5f071a9d1f404867857fcc
runc version: 54296cf40ad8143b62dbcaa1d90e520a2136ddfe
init version: 949e6fa
Security Options:
 seccomp
  Profile: default
Kernel Version: 3.10.0-693.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 992.4 MiB
Name: localhost.localdomain
ID: LTCP:QTVE:I6GV:AJD6:6EEP:PFAR:QCDW:YTQB:GCGR:YFQR:K46W:B4RS
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
```

3）安装 Rancher。

➢ 配置私有仓库。

```
echo '{ "insecure-registries":["reg.a.lan:5000"] }' >> /etc/docker/daemon.json
```

重启服务 systemctl restart docker。

```
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 17.03.2-ce
Storage Driver: overlay
 Backing Filesystem: xfs
 Supports d_type: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 4ab9917febca54791c5f071a9d1f404867857fcc
runc version: 54296cf40ad8143b62dbcaa1d90e520a2136ddfe
init version: 949e6fa
Security Options:
 seccomp
  Profile: default
Kernel Version: 3.10.0-693.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 992.4 MiB
Name: localhost.localdomain
ID: LTCP:QTVE:I6GV:AJD6:6EEP:PFAR:QCDW:YTQB:GCGR:YFQR:K46W:B4RS
```

```
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
 reg.a.lan:5000
 127.0.0.0/8
Live Restore Enabled: false
```

Rancher 相关的包有如下这些，需要自己联网拉取，然后制作离线安装。

```
rancher/healthcheck:v0.3.3
rancher/scheduler:v0.8.3
rancher/net:v0.13.7
rancher/dns:v0.15.3
rancher/net:holder
rancher/network-manager:v0.7.19
rancher/metadata:v0.9.5
rancher/agent:v1.2.9
rancher/server:stable
rancher/lb-service-haproxy:v0.7.15
```

打包 Rancher Server 及各组件镜像。

```
docker pull rancher/agent:v1.2.9
docker save rancher/agent:v1.2.9 > agent129.tar
```

导入镜像。登录 docker login 私有仓库 IP：端口。

```
docker load -i agent129.tar
docker tag rancher/agent:v1.2.9 reg.a.lan:5000/rancher/agent:v1.2.9
docker push reg.a.lan:5000/rancher/agent:v1.2.9
```

按照以上方式，将所有 Rancher image 导入私有镜像仓库，查看镜像仓库包，如图 7-6 所示。

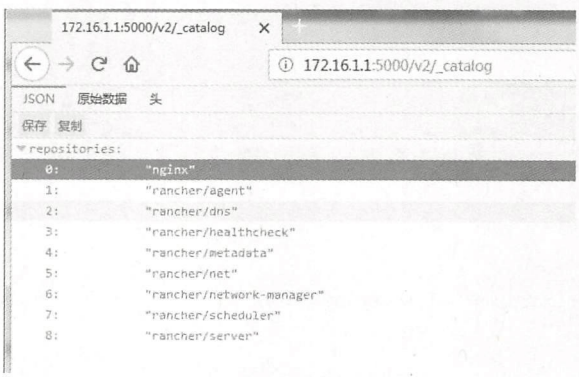


图 7-6　查看镜像仓库包

配置 Rancher 环境，在安装的主机上需要登录私有仓库，如果线上使用，可以把数据库挂载到本地硬盘上，则参数为-v /data:/var/lib/mysql。

```
docker run -d --restart=unless-stopped -p 8080:8080  reg.a.lan:5000/rancher/server:stable
Unable to find image 'reg.a.lan:5000/rancher/server:stable' locally
stable: Pulling from rancher/server
61866a02e134: Pull complete
5d9e73a9ac54: Pull complete
801ce3b4bd71: Pull complete
7429d79414dc: Pull complete
893dcd7c9449: Pull complete
Digest: sha256:d22b51c33a8500829c43a9c421d4ed5af9e14e633fe643fca9fd990dfd1e6cf3
Status: Downloaded newer image for reg.a.lan:5000/rancher/server:stable
5bf638cd0cd887244518799180001c1355a68d7a8284f1011e0dedcc4e9e4d770
```

安装完成后，可以在浏览器中打开管理页面，登录 UI，单击【系统管理】/【系统设置】/【高级设置】，将“名称”设置为 registry.default，“设置值”为 reg.a.lan:5000，如图 7-7 所示。

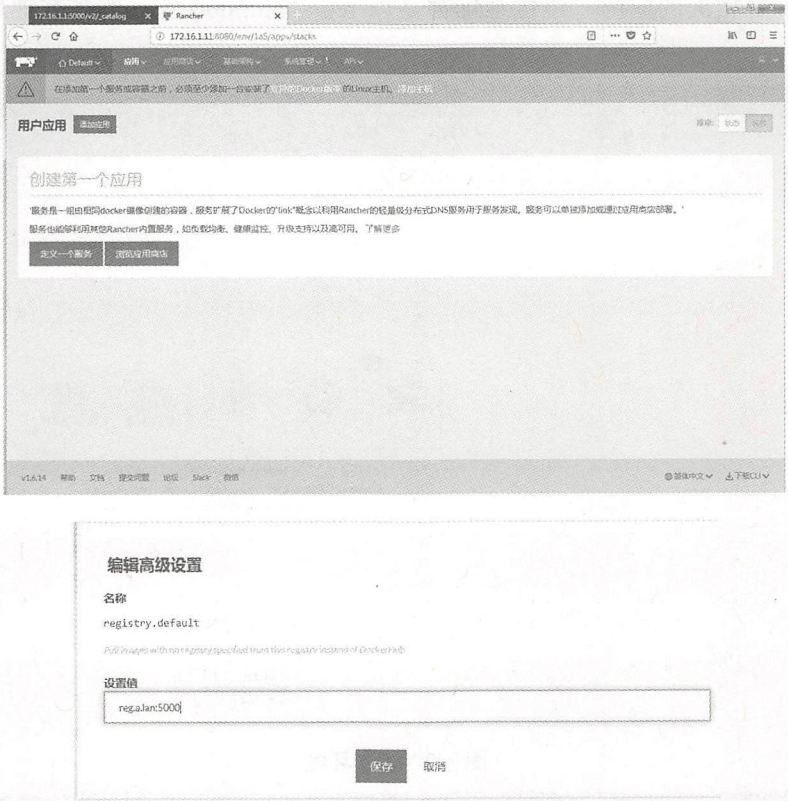


图 7-7　设置默认镜像库地址

配置私有镜像库，单击【基础架构】/【镜像库】，“添加镜像库”选择 Custom，如图 7-8 所示。

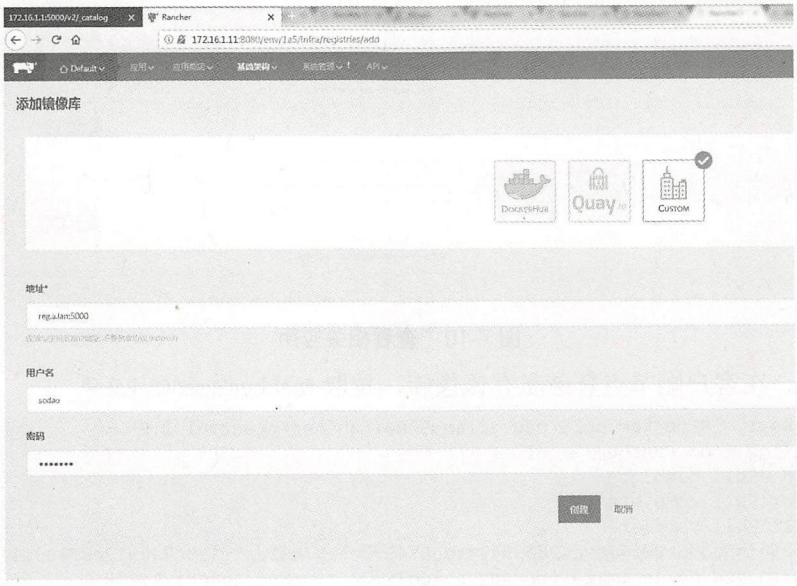


图 7-8　配置私有镜像库

添加一个环境，选择左上角的【环境管理】/【添加环境】，将“名称”设置为 myrancher，如图 7-9 所示。“环境模板”选择 Cattle，创建并切换到新创建的环境中。可以单击【应用】/【基础设施】查看相关的应用，镜像地址都是以 reg.a.lan:5000 开头的私有地址，如图 7-10 所示。

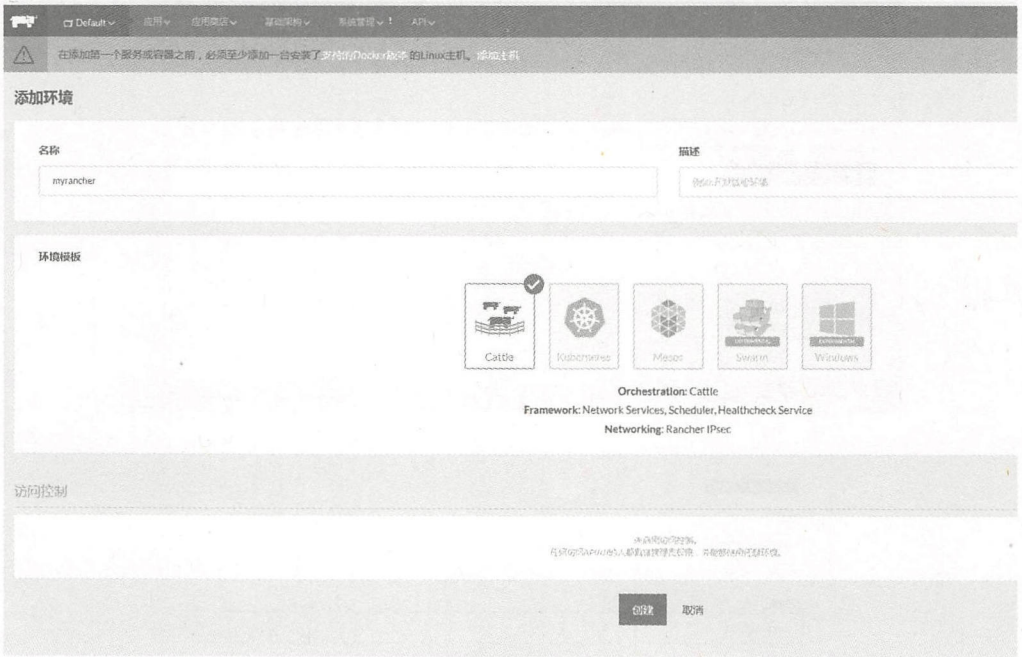


图 7-9　添加环境

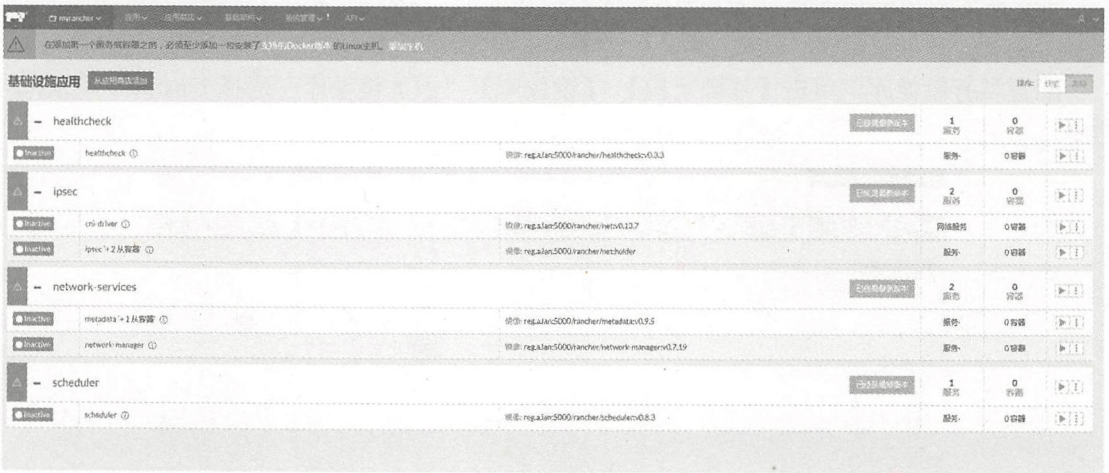


图 7-10　查看相关应用

添加主机，在客户端节点登录私有镜像库，拉取 rancher/agent:v1.2.9。

```
[root@localhost ~]# docker pull reg.a.lan:5000/rancher/agent:v1.2.9
v1.2.9: Pulling from rancher/agent
8339a52afd18: Pull complete
6a710864a9fc: Pull complete
d0ac3b234321: Pull complete
Digest: sha256:b5bcd6a00147ead428098494cdb8345f0efab7b02db1f130c9a2512e698e3111
Status: Downloaded newer image for reg.a.lan:5000/rancher/agent:v1.2.9
```

打标签为 rancher/agent:v1.2.9，后面添加主机时会用到。

```
[root@localhost ~]# docker tag reg.a.lan:5000/rancher/agent:v1.2.9 rancher/agent:v1.2.9
[root@localhost ~]# docker images
REPOSITORY                          TAG          IMAGE ID          CREATED          SIZE
reg.a.lan:5000/rancher/server       stable       d63b9b4bd205      6 months ago     1.08 GB
rancher/agent                       v1.2.9       34a453d374b9      6 months ago     237 MB
reg.a.lan:5000/rancher/agent        v1.2.9       34a453d374b9      6 months ago     237 MB
```

回到 Rancher UI 界面，选择添加主机，选择 Custom，填写将要添加的主机 IP 地址，如图 7-11 所示。

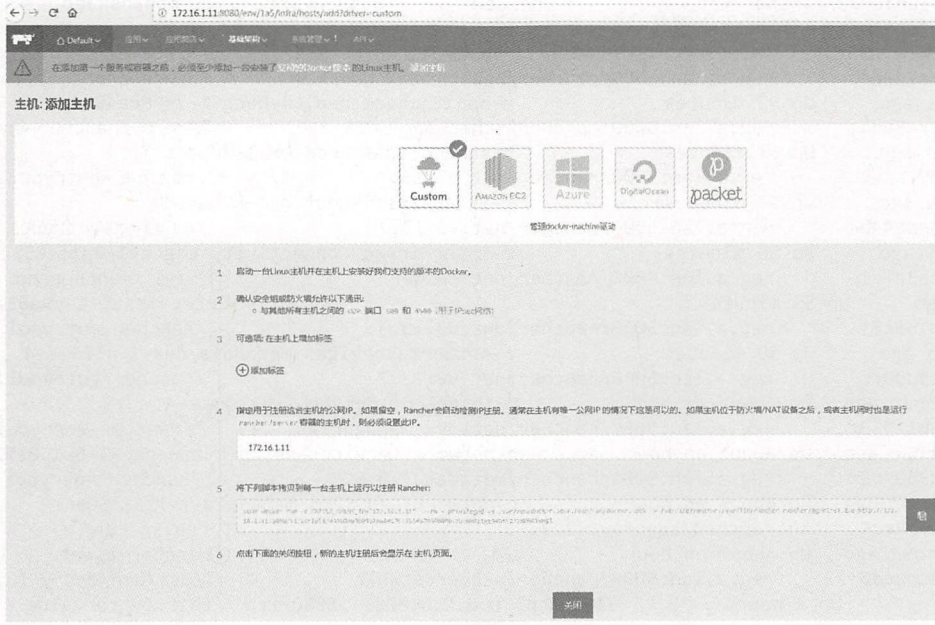


图 7-11 填写主机 IP 地址

将脚本复制到主机上运行，注册 Rancher。

```
[root@localhost ~]# sudo docker run -e CATTLE_AGENT_IP="172.16.1.11"  --rm --privileged -v /var/
run/docker.sock:/var/run/docker.sock  -v  /var/lib/rancher:/var/lib/rancher  rancher/agent:  v1.2.9
http://172.16.1.11:8080/v1/scripts/CA9946C8D0B8CE39F70F:1514678400000:ZUPJoDB3EfyAqH9sgkHRxUcGCyM
INFO: Running Agent Registration Process, CATTLE_URL=http://172.16.1.11:8080/v1
INFO: Attempting to connect to: http://172.16.1.11:8080/v1
INFO: http://172.16.1.11:8080/v1 is accessible
INFO: Inspecting host capabilities
INFO: Boot2Docker: false
INFO: Host writable: true
INFO: Token: xxxxxxxx
INFO: Running registration
INFO: Printing Environment
INFO: ENV: CATTLE_ACCESS_KEY=6F604AE48931B3425F94
INFO: ENV: CATTLE_AGENT_IP=172.16.1.11
INFO: ENV: CATTLE_HOME=/var/lib/cattle
INFO: ENV: CATTLE_REGISTRATION_ACCESS_KEY=registrationToken
INFO: ENV: CATTLE_REGISTRATION_SECRET_KEY=xxxxxxx
INFO: ENV: CATTLE_SECRET_KEY=xxxxxxx
INFO: ENV: CATTLE_URL=http://172.16.1.11:8080/v1
INFO: ENV: DETECTED_CATTLE_AGENT_IP=172.17.0.1
INFO: ENV: RANCHER_AGENT_IMAGE=rancher/agent:v1.2.9
INFO: Launched Rancher Agent: 58eb0a7e231f26723316120d2167e921bc642f7f6b26220df43411f91162c638
```

从日志可以看到从私有镜像库中获取镜像，此处日志较多，不进行展示。

查看客户端镜像列表和运行的容器。

```
[root@localhost ~]# docker images
```

```
REPOSITORY                                TAG          IMAGE ID       CREATED          SIZE
reg.a.lan:5000/rancher/server             stable       d63b9b4bd205   6 months ago     1.08 GB
reg.a.lan:5000/rancher/agent              v1.2.9       34a453d374b9   6 months ago     237 MB
rancher/agent                             v1.2.9       34a453d374b9   6 months ago     237 MB
reg.a.lan:5000/rancher/scheduler          v0.8.3       3e640a41799a   6 months ago     242 MB
reg.a.lan:5000/rancher/net                v0.13.7      1d3351fae706   6 months ago     310 MB
reg.a.lan:5000/rancher/network-manager    v0.7.19      87e7ab1c6276   7 months ago     256 MB
reg.a.lan:5000/rancher/metadata           v0.9.5       bd33f8c865b1   8 months ago     251 MB
reg.a.lan:5000/rancher/dns                v0.15.3      2779a18358f2   10 months ago    240 MB
reg.a.lan:5000/rancher/healthcheck        v0.3.3       14de771cc178   10 months ago    385 MB
reg.a.lan:5000/rancher/net                holder       665d9f6e8cc1   15 months ago    267 MB
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED             STATUS              PORTS               NAMES
3e6f41a7d0e4        reg.a.lan:5000/rancher/healthcheck:v0.3.3   "/.r/r /rancher-en..."
57 minutes ago      Up 57 minutes                          r-healthcheck-healthcheck-1-73f8ea40
af53737e8ed4        reg.a.lan:5000/rancher/scheduler:v0.8.3     "/.r/r /rancher-en..."
57 minutes ago      Up 57 minutes                          r-scheduler-scheduler-1-dd8acc17
7d5c2d9fa4eb        reg.a.lan:5000/rancher/net:v0.13.7          "/rancher-entrypoi..."
58 minutes ago      Up 57 minutes                          r-ipsec-ipsec-router-1-82cadbdb
aade43ede44b        reg.a.lan:5000/rancher/net:v0.13.7          "/rancher-entrypoi..."
58 minutes ago      Up 58 minutes                          r-ipsec-ipsec-connectivity-check-1-0a1f1055
e316deb3afd1        reg.a.lan:5000/rancher/net:holder           "/.r/r /rancher-en..."   59
minutes ago      Up 59 minutes                                 r-ipsec-ipsec-1-a9a03e40
635a231b5c81        reg.a.lan:5000/rancher/dns:v0.15.3          "/rancher-entrypoi..."
59 minutes ago      Up 59 minutes                          r-network-services-metadata-dns-1-450518e4
869a21dd90fb        reg.a.lan:5000/rancher/net:v0.13.7          "/rancher-entrypoi..."
About an hour ago   Up About an hour                       r-ipsec-cni-driver-1-a33f94a3
547b8ab12553        reg.a.lan:5000/rancher/network-manager:v0.7.19   "/rancher-entrypoi..."
About an hour ago   Up About an hour                       r-network-services-network-manager-1-6e1508f1
bdd80d3b03b8        reg.a.lan:5000/rancher/metadata:v0.9.5      "/rancher-entrypoi..."
About an hour ago   Up About an hour                       r-network-services-metadata-1-f161c50e
922e5bfc8ec6        rancher/agent:v1.2.9                                 "/run.sh run"
About an hour ago   Up About an hour                                     rancher-agent
5bf638cd0cd8        reg.a.lan:5000/rancher/server:stable        "/usr/bin/entry /u..."   2
hours ago        Up 2 hours          3306/tcp, 0.0.0.0:8080->8080/tcp   thirsty_torvalds
```

查看 UI 页面，单击【基础架构】/【主机】，容器都已正常运行，如图 7-12、图 7-13 所示。

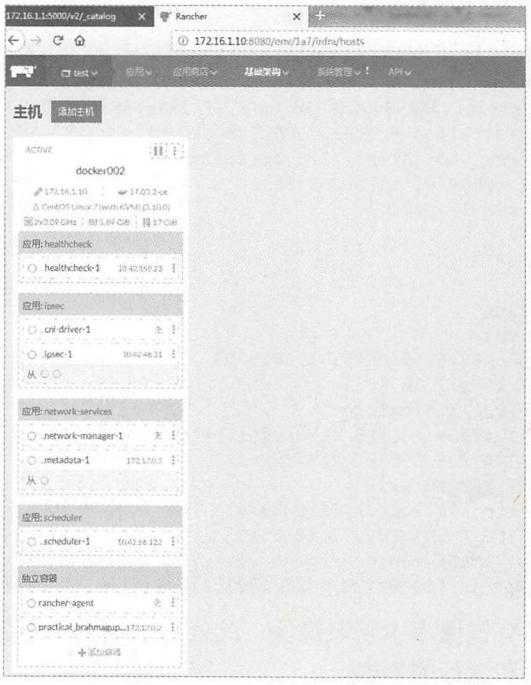


图 7-12　容器正常运行

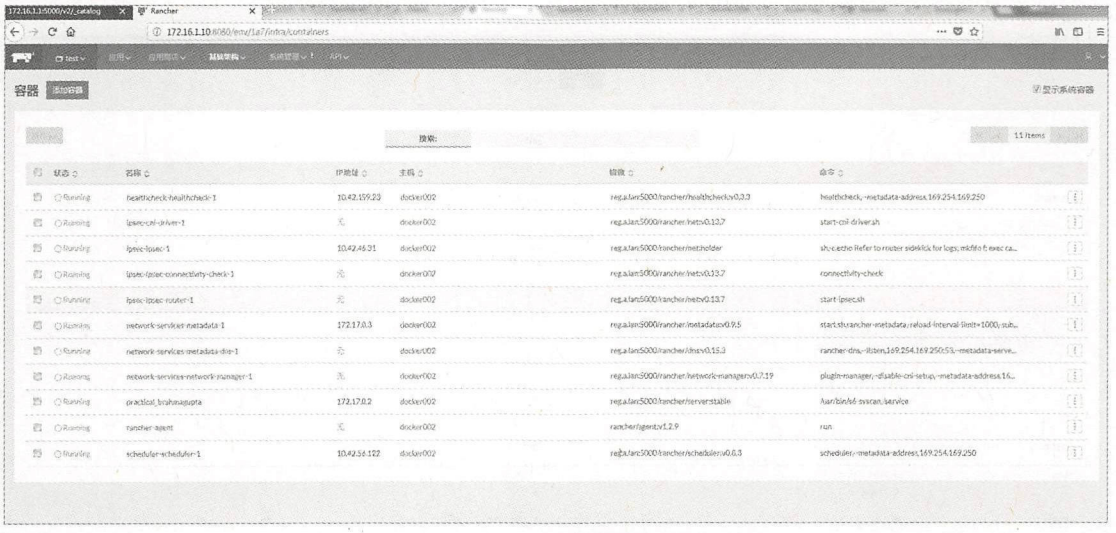


图 7-13　容器运行状况

应用创建和使用。单击【应用】/【全部】/【添加应用】，在“名称”中输入 myapp，这里支持直接上传 docker-compose.yml，如图 7-14 所示。

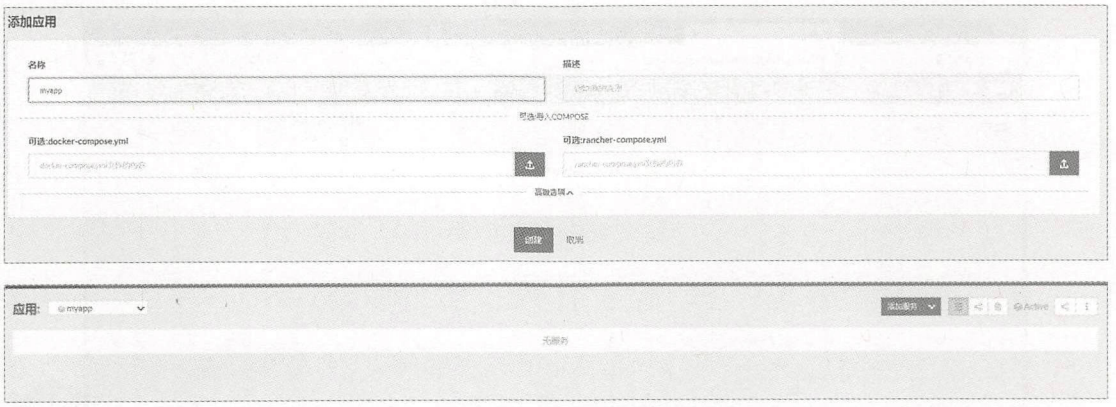


图 7-14　应用创建和使用

添加一个 Nginx 服务网络模式，此处选择主机创建完毕后直接使用宿主机访问 Nginx，如图 7-15 所示。

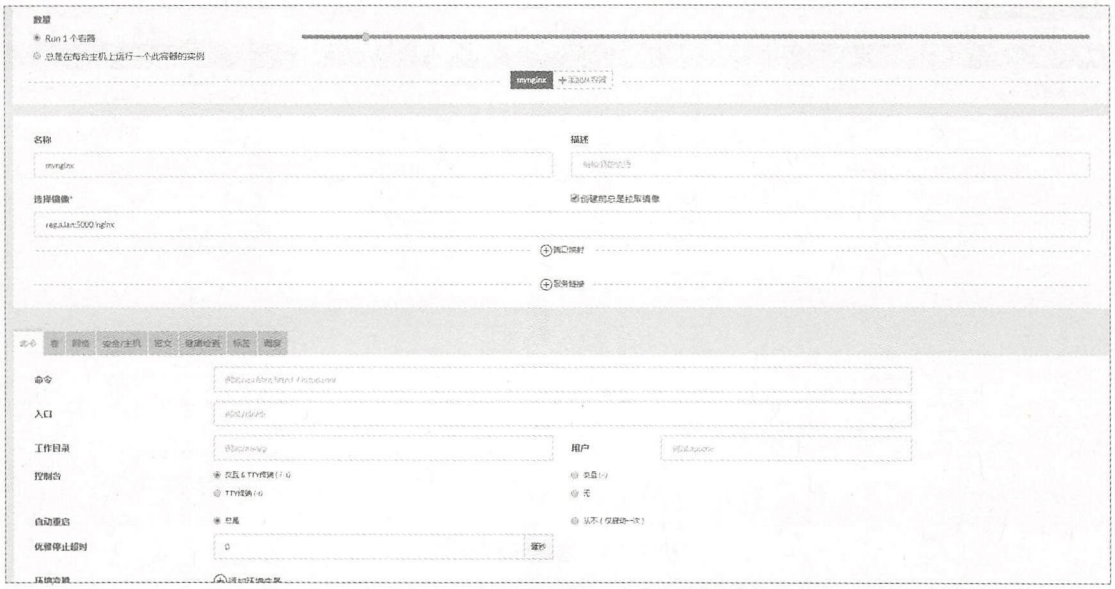


图 7-15　添加一个 Nginx 服务网络模式

创建完成后，显示 Active 创建成功，如图 7-16 所示。

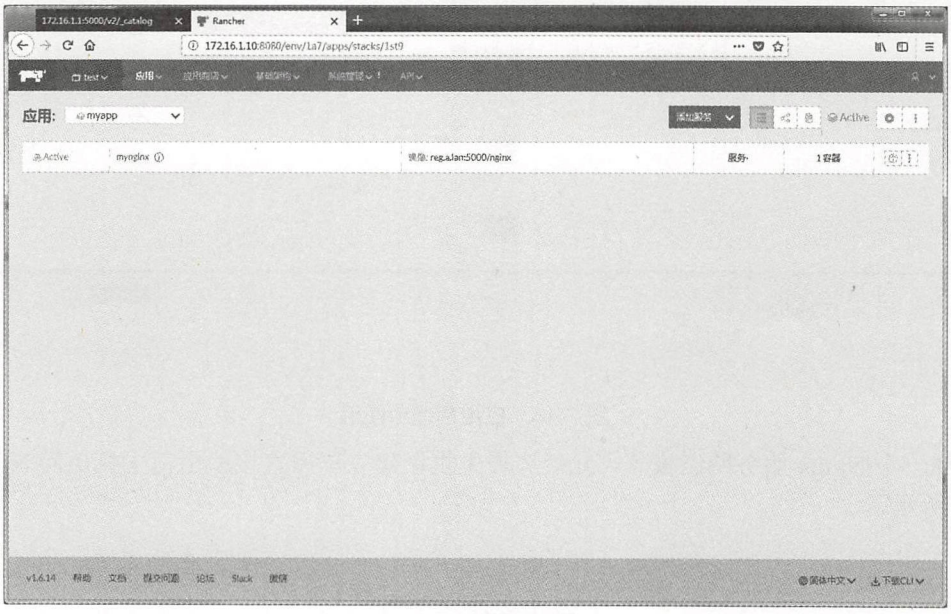


图 7-16　创建完成

访问宿主机房的地址为 http://172.16.1.10/，到这里整个应用就创建好了，如图 7-17 所示。

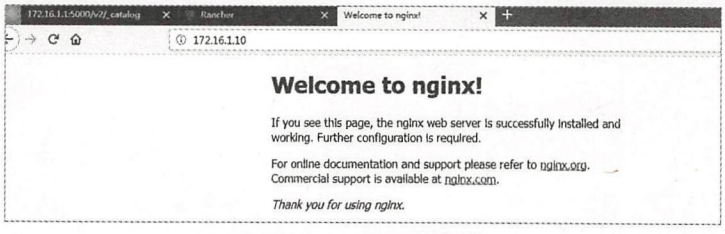


图 7-17　应用创建完成

负载均衡与服务扩展。一个服务单点运行是不可取的，万一宕机则无法提供服务，Rancher可以很方便地对服务进行扩展，只需要在界面上操作即可，如图 7-18 所示。

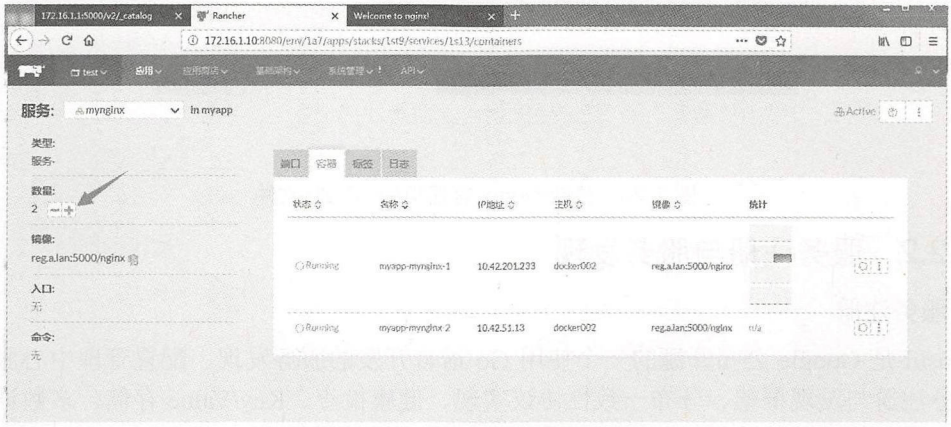


图 7-18 负载均衡与服务扩展

添加一个负载均衡服务对外提供服务，自带的负载均衡服务是 HAProxy，如图 7-19 所示。

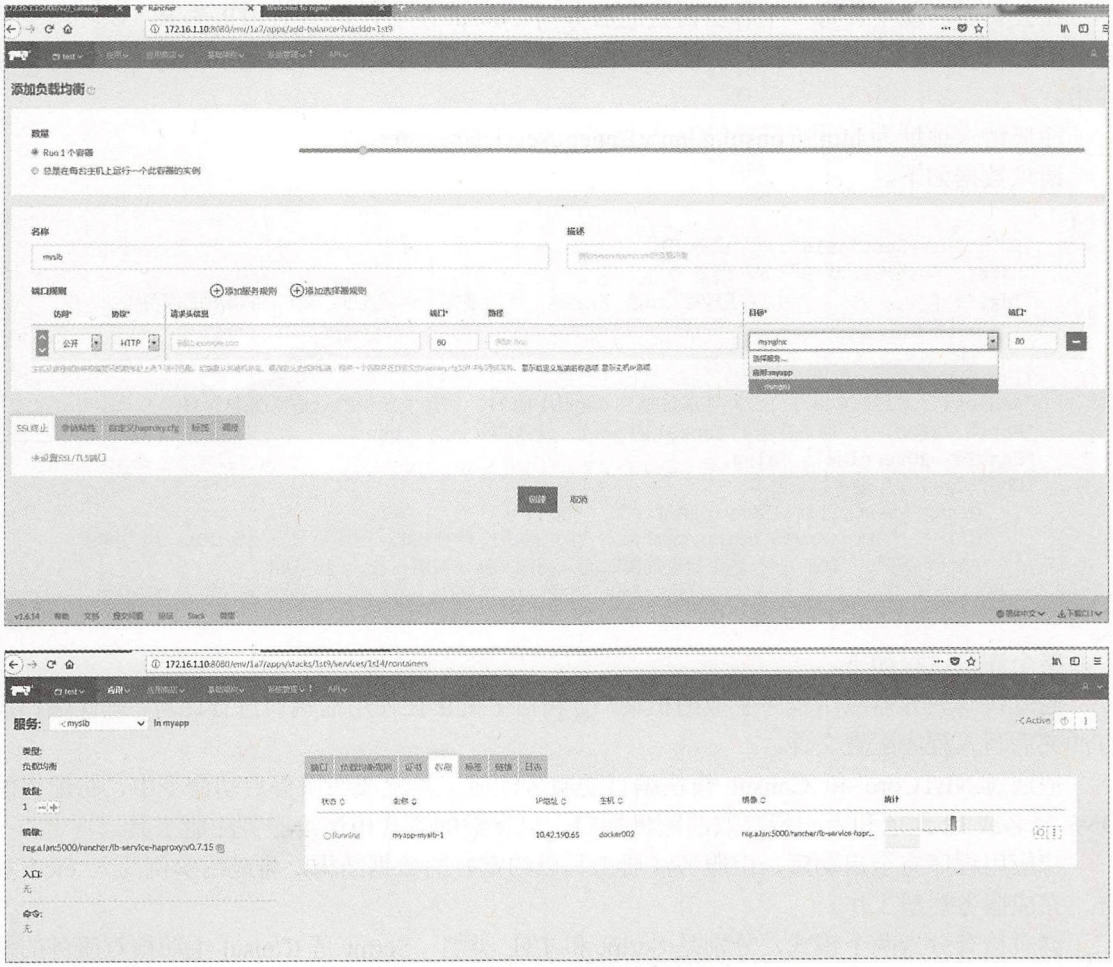


图 7-19 添加一个负载均衡服务

为了区分负载均衡效果，直接修改 Nginx 容器里的 HTML 文件，如图 7-20 所示。

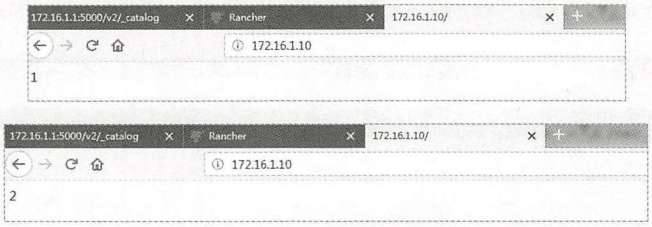


图 7-20　修改 Nginx 容器里的 HTML 文件

7.2.2　服务注册与服务发现

1. 服务注册

Consul 是 Google 公司开源的一个使用 Go 语言开发的服务发现、配置管理中心服务。内置了服务注册与发现框架、分布一致性协议实现、健康检查、Key/Value 存储、多数据中心方案，不再需要依赖其他工具（比如 ZooKeeper 等）。服务部署简单，只有一个可运行的二进制包。每个节点都需要运行 Agent，有两种运行模式 Server 和 Client。每个数据中心需要 3 或 5 个 Server 节点以保证数据安全，同时保证 Server-leader 的选举能够正确地进行。

在公司的环境中，每一个容器内置的服务向 Consul 注册，使用 HTTP 请求，以环境变量的 DNS 解析的域名或者本身容器的 host+port 作为唯一通道，保存到 consul key-value 数据仓库中。

注册请求地址为 http://consul.g.lan/v1/agent/service/register。

请求数据如下。

```
{
    "ID": "userServiceId", // 服务 ID
    "Name": "userService", // 服务名
    "Tags": [              // 服务的 tag，自定义，可以根据 tag 区分同一个服务器名的服务
        "primary",
        "v1"
    ],
    "Address": "127.0.0.1", // 服务注册到 Consul 的 IP，服务发现指的就是发现这个 IP
    "Port": 8000, // 服务注册到 Consul 的 port，发现的就是这个 port
    "EnableTagOverride": false,
    "Check": { // 健康检查部分
        "DeregisterCriticalServiceAfter": "90m",
        "HTTP": "http://www.baidu.com", // 指定健康检查的 URL，调用后只要返回 20x，均为健康
        "Interval": "10s" // 健康检查间隔时间，每隔 10s，调用一次上门的 URL
    }
}
```

整个注册过程如下。

通过代码为容器应用程序生成随机端口，和宿主机正在使用的端口进行比对，确保端口没有冲突后写入程序配置文件。

把通过.NETCore 和 Consul 模块编写的服务注册工具集成在服务启动命令中，将配置的 DNS 域名或 IP 地址和上一步获取的随机端口，以参数的方式传递给服务注册工具。

待应用程序完全启动后，由服务注册工具以约定好的数据结构，将应用实例写入 etcd 集群，完成服务注册工作。

健康检查分为两个模式，分别是 Script 和 TTL 类型。Script 是 Consul 主动检查服务的健康状况，TTL 是服务主动向 Consul 报告自己的状况。

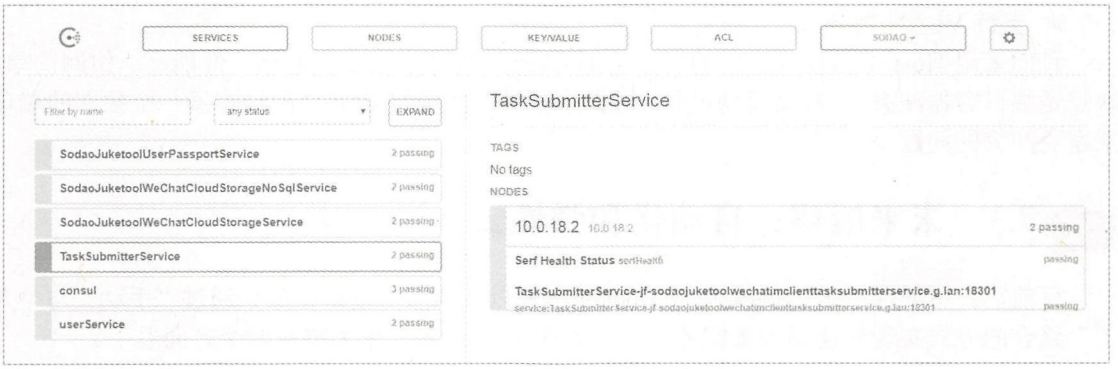


图 7-21　注册中心示意图

2. 服务发现

利用 Consul 提供的 HTTP 请求接口，根据服务唯一标识获取可用节点信息。

请求地址为 http://consul.g.lan/v1/catalog/service/{服务唯一标识}。

返回的响应数据如下所示。

```
[
    {
        "ID": "009c0dec-fdeb-768f-6ee5-b54e687d3e4d",
        "Node": "10.0.12.10",
        "Address": "10.0.12.10",
        "Datacenter": "dc1",
        "TaggedAddresses": {
            "lan": "10.0.12.10",
            "wan": "10.0.12.10"
        },
        "NodeMeta": {
            "consul-network-segment": ""
        },
        "ServiceID": "SodaoJuketoolWeChatCloudStorageService-10.0.120.21:18201",
        "ServiceName": "SodaoJuketoolWeChatCloudStorageService",
        "ServiceTags": [],
        "ServiceAddress": "10.0.120.21",
        "ServicePort": 18201,
        "ServiceEnableTagOverride": false,
        "CreateIndex": 601566,
        "ModifyIndex": 601566
    }
]
```

7.2.3　Docker 网络与通信解决方案

Docker 的网络模式已经在前文做了介绍，这里不再赘述。但是由于 Docker 自身的网络功能比较简单，不能满足很多复杂的应用场景，因此有很多开源项目用来改善 Docker 的网络功能，例如网络配置工具 Pipework，下面简单介绍一下。

Pipework 是一个简单易用的 Docker 容器网络配置工具，由 200 多行 shell 脚本实现。通过使用 ip、brctl、ovs-vsctl 等命令为 Docker 容器配置自定义的网桥、网卡、路由等。其有如下功能：

- 支持使用自定义的 Libux Bridge、veth pair 为容器提供通信。
- 支持使用 macvlan 设备将容器连接到本地网络。
- 支持 DHCP 获取容器的 IP。
- 支持 Open vSwitch。

➢ 支持 VLAN 划分。

我们采用 Host 主机模式，它的优点是 Docker 原生共享宿主机网络、性能高、组网简单，缺点是端口容器冲突，在容器启动过程中会执行脚本检查宿主机并分配给容器一个独立的端口，来避免冲突的问题。

7.3 未来展望：自动化和弹性云

目前，公司平台基于 ZStack 集成 Docker 容器以及微服务基础框架，通过“手动”与“人工”结合的方式实现快速添加微服务节点，如图 7-23 所示，未来将从两个方面着手。

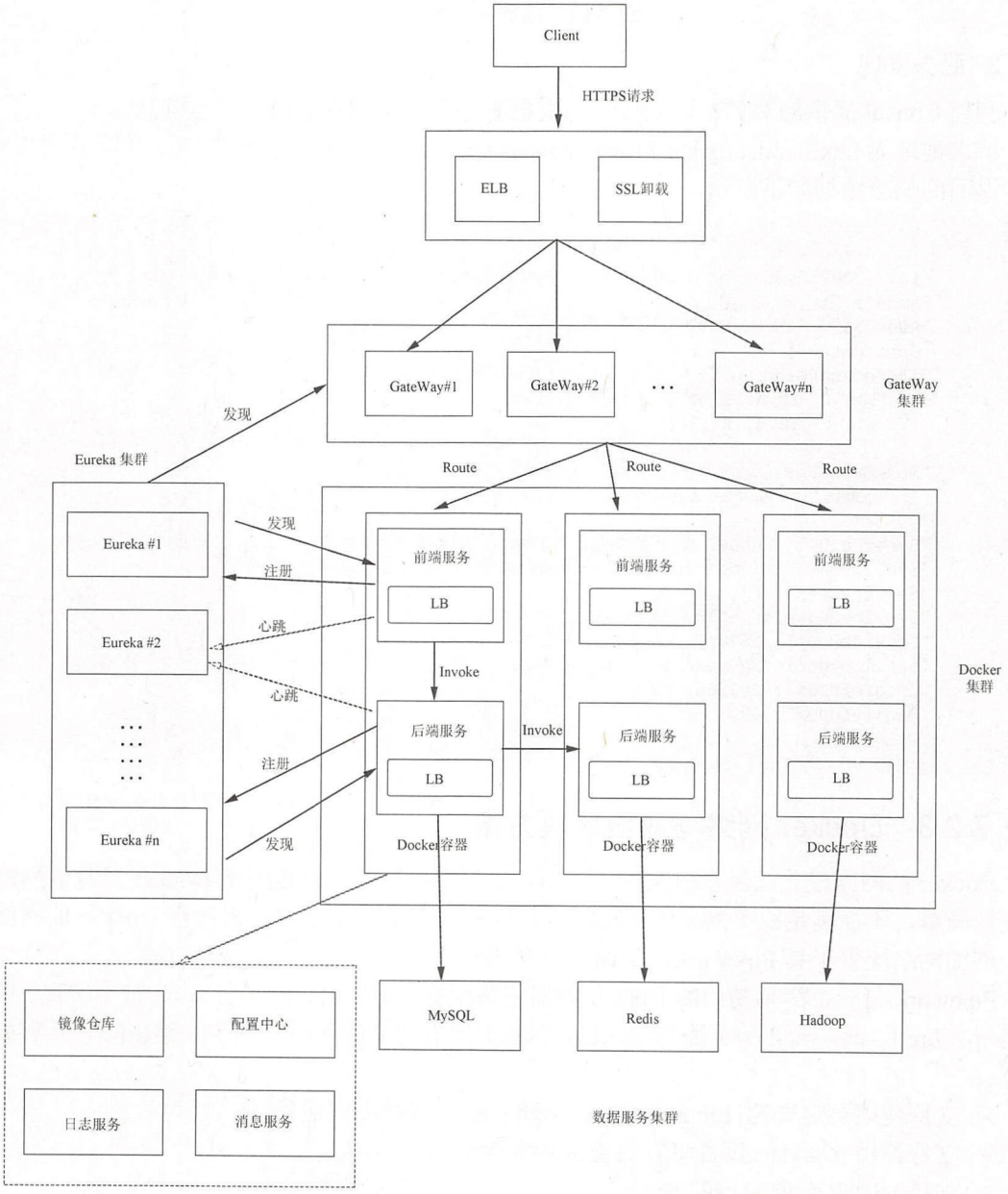


图 7-23 系统架构

7.3.1 自动化

微服务天然需要自动化工具来降低发布部署的出错率，提高产品迭代的效率。如图 7-24 所示，目前搜道系统应用迭代的最终步骤流转，实现自动化发布和部署。

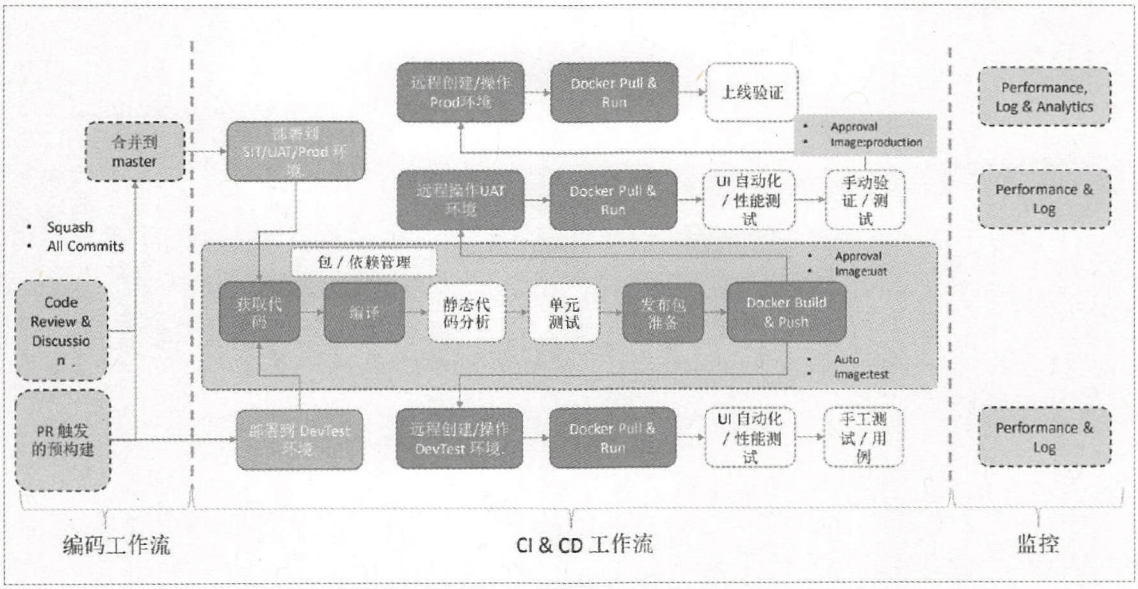


图 7-24 CI/CD 工作流示意图

1. 持续集成

检测代码提交状态，对代码进行持续集成，在集成过程中执行单元测试，用代码 Sonar 和安全工具进行静态扫描，将结果通知给开发人员的同时部署集成环境，部署成功后触发自动化测试。

2. 持续部署

快速部署，一个包可以快速地部署到任何一个地方。

7.3.2 弹性云

利用 Rancher 作为容器云平台的编排工具，在对应用的容器实例进行统一的编排调度时，配合 Docker-Compose 组件，可以在同一时间对多台宿主机执行调度操作。同时，在服务访问出现峰值和低谷时，利用特有的 rancher-compose.yml 文件调用“SCALE”特性，对应用集群执行动态扩容和缩容，让应用按需求处理不同的请求。

本章作者：徐常雨。

CHAPTER

8

第8章 纵横新创的容器化实践

当今互联网行业发展迅速，产品架构越来越复杂，导致搭建开发测试环境越来越困难。当项目开发完成后，一般会在测试环境上运行，供开发部门调错和测试部门测试。对于具有一定业务规模的公司，有几十个甚至上百个应用服务，每个服务分别占用一个目录，配置过程烦琐，且无法集中管理。

为了提高开发与测试环节的工作效率，我们用 Rancher 搭建了一套容器云平台，并结合 Jenkins 实现持续集成与持续交付。

自从将开发测试环境迁入容器云后，大大提高了环境构建和应用的部署速度，可以随时随地将环境和应用一起切换到某个版本状态。将开发人员和测试人员从琐碎的构建部署工作中解放出来，大大提高了工作效率。

本文将重点介绍开发测试环境的 Rancher 容器云的安装与配置过程及安装过程中遇到的问题，并探讨未来生产环境容器云的路线与规划。

8.1　背景介绍

纵横新创旗下的纵横理财是一家成长迅速的互联网金融理财平台。随着业务量的提升、存管接入、各种互联网金融合规改造的开展，原基于 Spring MVC 的单体式架构（如图 8-1 所示）已经不能满足业务发展的要求，为此在 2018 年启动了系统重构。新的系统架构基于 Spring Cloud 的微服务架构搭建，如图 8-2 所示。

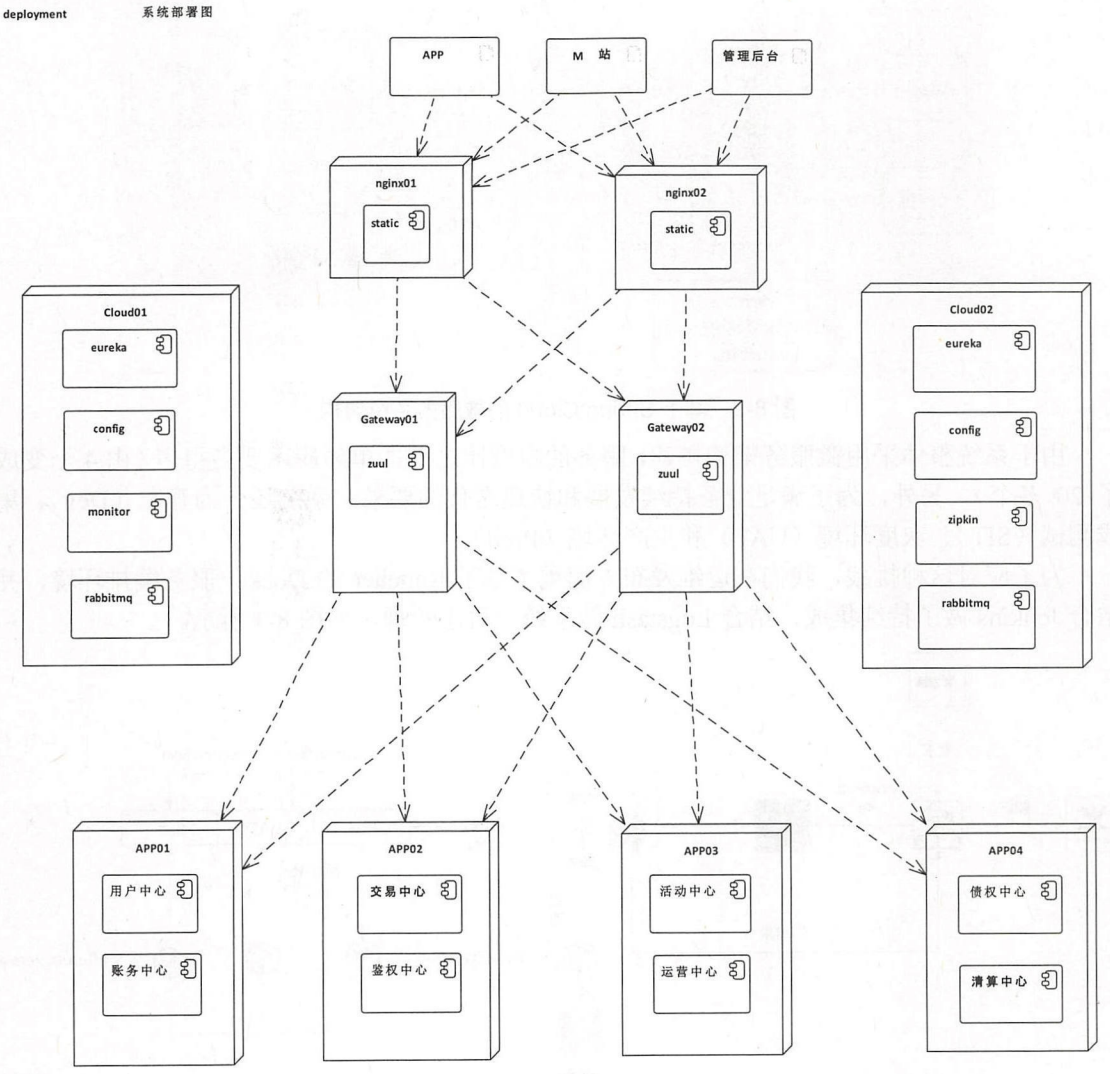


图 8-1　单体式架构

- 外部或者内部的非 Spring Cloud 项目统一通过 API 网关（Zuul）访问内部服务。
- 网关接收到请求后，从注册中心（Eureka）获取可用服务。
- 由 Ribbon 进行均衡负载后，分发到后端的具体实例。
- 微服务之间通过 Feign 进行通信处理业务。
- 内部服务调用采用 Hystrix 负责处理服务超时熔断。
- Monitor 用 Turbine 监控服务间的调用和熔断相关指标，并用 Sleuth 做调用链监控。

- 用基于 RabbitMQ 的 Spring Cloud Stream 实现消息处理。
- Nginx 在给 Zuul 做负载均衡的同时，扮演静态资源服务器的角色。

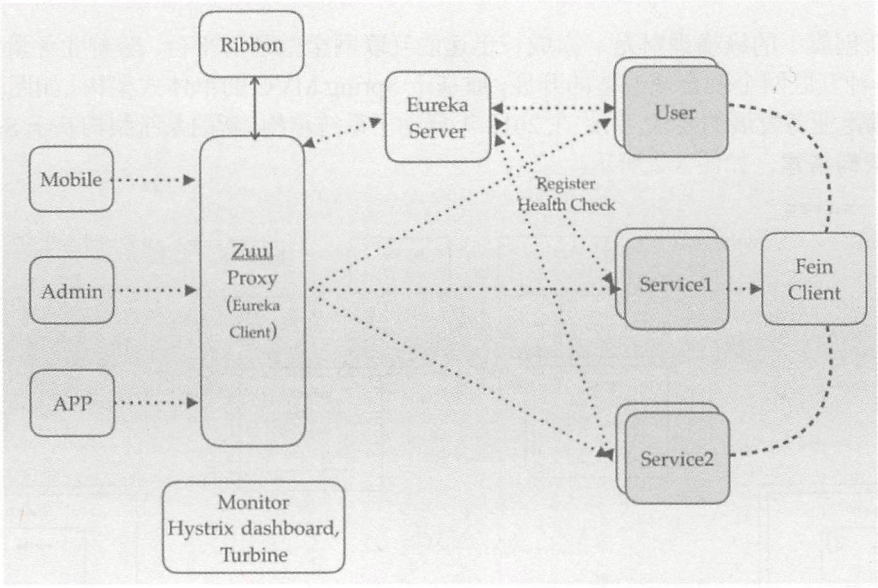


图 8-2　基于 Spring Cloud 的微服务系统架构

由于系统整体采用微服务架构搭建，服务的粒度比之前的单体应用要多得多（由 4 个变成了 20 多个）。另外，为了满足业务持续发展和快速迭代的要求，系统还分为开发（Dev）、集成测试（SIT）、灰度环境（UAT）和生产环境（Prod）。

为了应对这种挑战，我们在运维发布方面用了基于 Rancher 的 Docker 服务编排环境，并结合 Jenkins 做了持续集成，结合 Logstash 做了统一日志管理，如图 8-3 所示。

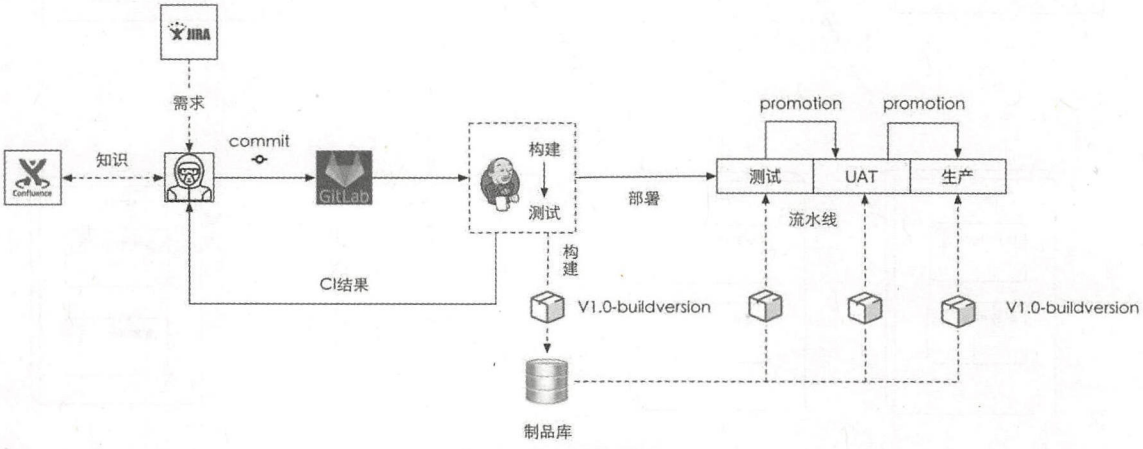


图 8-3　持续集成步骤

8.2　Rancher 介绍

Rancher 是一个开源的企业级容器管理平台，提供了在生产环境中使用的管理 Docker 和 Kubernetes 的全栈化容器部署与管理平台。通过 Rancher，企业再也不必自己使用一系列的开源软件从头搭建容器服务平台。

Rancher 由基础设施编排、应用商店、容器编排与调度、企业级权限管理四个部分组成。

8.2.1　基础设施编排

Rancher 可以使用任何公有云或私有云的 Linux 主机资源。Linux 主机可以是虚拟机，也可以是物理机。Rancher 仅需要主机有 CPU、内存、本地磁盘和网络资源。从 Rancher 的角度来说，一台云厂商提供的云主机和一台自己的物理机是一样的。

Rancher 为运行容器化的应用实现了一层灵活的基础设施服务。Rancher 的基础设施服务包括网络、存储、负载均衡、DNS 和安全模块。Rancher 的基础设施服务也是通过容器部署的，所以，同样 Rancher 的基础设施服务可以运行在任何 Linux 主机上。

8.2.2　应用商店

Rancher 的用户可以在应用商店里一键部署由多个容器组成的应用。用户可以管理应用，并且可以在应用有新的可用版本时进行自动化升级。Rancher 提供了一个由 Rancher 社区维护的应用商店，包括一系列的流行应用。Rancher 的用户也可以创建自己的私有应用商店。

8.2.3　容器编排与调度

很多用户都会选择使用容器编排调度框架运行容器化应用。Rancher 包含了当前全部主流的编排调度引擎，例如 Docker Swarm、Kubernetes 和 Mesos。同一个用户可以创建 Swarm 或 Kubernetes 集群，并且可以使用原生的 Swarm 或 Kubernetes 工具管理应用。

除了 Swarm、Kubernetes 和 Mesos，Rancher 还支持 Cattle 容器编排调度引擎。Cattle 被广泛用于编排 Rancher 的基础设施服务，以及 Swarm 集群、Kubernetes 集群和 Mesos 集群的配置、管理与升级。

8.2.4　企业级权限管理

Rancher 支持灵活的插件式的用户认证，支持 Active Directory、LDAP、GitHub 等认证方式。Rancher 支持环境级别的、基于角色的访问控制（RBAC），可以通过角色配置某个用户或用户组对开发环境或生产环境的访问权限。

8.3　Docker 构件库配置

使用 Docker 进行开发测试的配置与部署需要用到 Docker 镜像库，镜像库可以用 Docker-Registry 搭建，上传到外部的 Docker Hub 上面。考虑到性能和安全性方面的原因，一般要在研发团队的局域网里部署一套 Docker 镜像环境。

另外，在现有的软件开发团队里，一般是用 Maven 进行构建的。在用这些工具进行编译的时候，往往需要一个构件库与之配套。如果能够把研发的 Maven 构件库和 Docker 镜像库结合在一起，将极大地方便研发团队的环境管理与维护。

Nexus Repository Manager OSS 3 不但支持 Maven 构件，而且支持 Docker 镜像。OSS3 支持的构件库类型有 Bower、Docker、Git LFS、Maven、npm、NuGet、PyPI、Ruby Gems、Yum Proxy/Host。

8.3.1 Nexus 3 安装

Nexus 3 支持 Docker 安装，可以用传统的方式进行安装。由于 Nexus 3 扮演 Docker 镜像库的角色，为了和 Docker 编排环境（Rancher）解耦，因此采用传统的安装模式。

Nexus 3 可以创建三种 Docker 仓库：docker（proxy）、代理和缓存远程仓库只能拉取；docker（hosted）、托管仓库、私有仓库可以推送和拉取；docker（group）将多个 proxy 和 hosted 仓库添加到一个组，只访问一个组地址即可，只能拉取。

安装配置 Java 环境，如图 8-10 所示。

wget http://download.oracle.com/otn-pub/java/jdk/8u181-b13/96a7b8442fe848ef90c96a2fad6ed6d1/jdk-8u181-linux-i586.tar.gz。

```
[root@renchar opt]# wget http://download.oracle.com/otn-pub/java/jdk/8u161-b12/2f38c3b165be4555a1fa6e98c45e0808/jdk-8u161-linux-x64.tar.gz
--2018-03-25 11:16:55--  http://download.oracle.com/otn-pub/java/jdk/8u161-b12/2f38c3b165be4555a1fa6e98c45e0808/jdk-8u161-linux-x64.tar.gz
正在解析主机 download.oracle.com (download.oracle.com)... 23.57.112.199
正在连接 download.oracle.com (download.oracle.com)|23.57.112.199|:80... 已连接。
已发出 HTTP 请求，正在等待回应... 302 Found
位置：http://111.1.62.31/files/216700000680F6BC/download.oracle.com/otn-pub/java/jdk/8u161-b12/2f38c3b165be4555a1fa6e98c45e0808/jdk-8u161-l
随至新的 URL]
--2018-03-25 11:16:55--  http://111.1.62.31/files/216700000680F6BC/download.oracle.com/otn-pub/java/jdk/8u161-b12/2f38c3b165be4555a1fa6e98c
nux-x64.tar.gz
正在连接 111.1.62.31:80... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度：189756259 (181M) [application/octet-stream]
正在保存至："jdk-8u161-linux-x64.tar.gz.1"
```

图 8-10 安装配置 Java 环境

下载 OSS 3 安装文件，如图 8-11 所示。

wget https://sonatype-download.global.ssl.fastly.net/repository/repositoryManager/3/nexus-3.9.0-01-unix.tar.gz。

```
[root@renchar opt]# wget https://sonatype-download.global.ssl.fastly.net/repository/repositoryManager/3/nexus-3.9.0-01-unix.tar.gz
--2018-03-25 11:20:14--  https://sonatype-download.global.ssl.fastly.net/repository/repositoryManager/3/nexus-3.9.0-01-unix.tar.gz
正在解析主机 sonatype-download.global.ssl.fastly.net (sonatype-download.global.ssl.fastly.net)... 151.101.41.194
正在连接 sonatype-download.global.ssl.fastly.net (sonatype-download.global.ssl.fastly.net)|151.101.41.194|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度：113125671 (108M) [application/x-gzip]
正在保存至："nexus-3.9.0-01-unix.tar.gz"

 2% [=>                                                                                                  ] 2,547,712
```

图 8-11 下载 OSS 3 安装文件

```
tar zxvf nexus-3.9.0-01-unix.tar.gz
cd nexus-3.9.0-01/bin
./nexus run
```

在浏览器里打开 http://192.168.1.107:8081，默认用户名及密码分别是 admin 和 admin123，如图 8-12 所示。

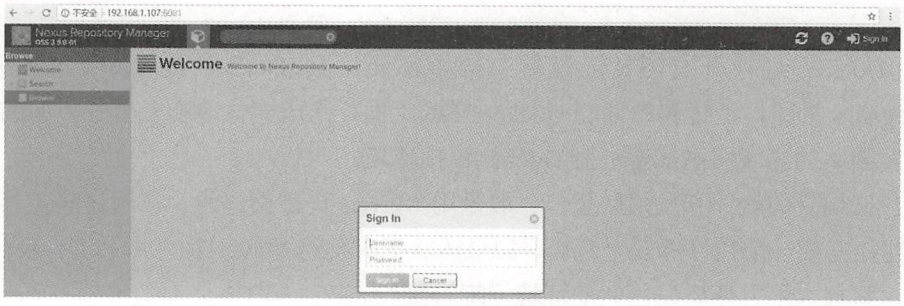


图 8-12 Nexus 3 登录界面

注意：如果浏览器打不开，则需要关闭或调整 Linux 的防火墙。要在 limits.conf 中把 Nexus 所在的用户的 nofile 调整到 65536。

8.3.2　Nexus 3 配置 Docker 镜像库

像部署一台 Maven 私服一样开始操作，在设置 Repositories 选项卡中选择 Create Repository，如图 8-13 所示。

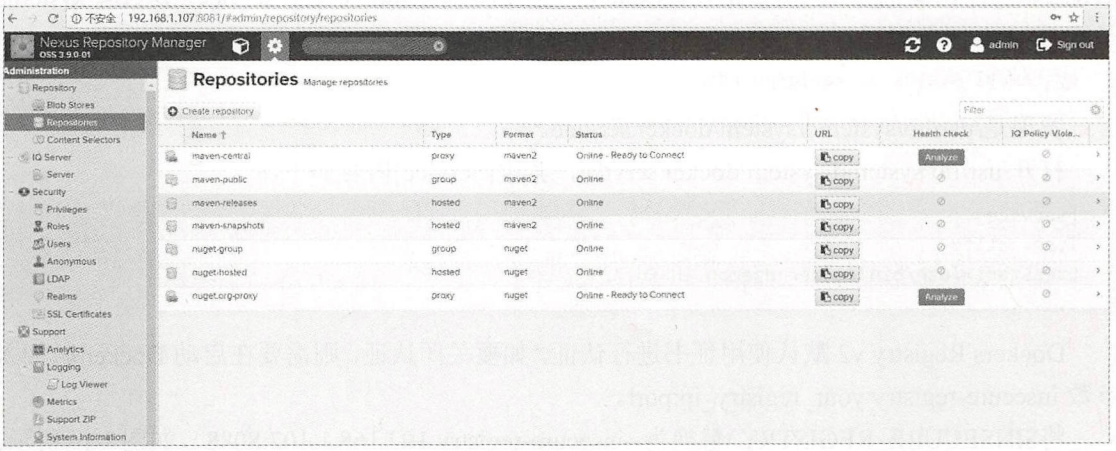


图 8-13　选择 Create repository

Docker 镜像仓库有三种类型，分别是 docker（group）、docker（hosted）和 docker（proxy）。hosted 表示本地存储，即同 Docker 官方仓库一样提供本地私服功能；proxy 表示提供代理其他仓库的类型，如 Docker 中央仓库；group 表示组类型，实质作用是组合多个仓库为一个地址。

因为目标是建立一个本地私服 Docker 镜像仓库，所以选择 docker（hosted），填写仓库名称、端口（如 8088）等信息后，单击 Create Repository 创建即可，如图 8-15 所示。

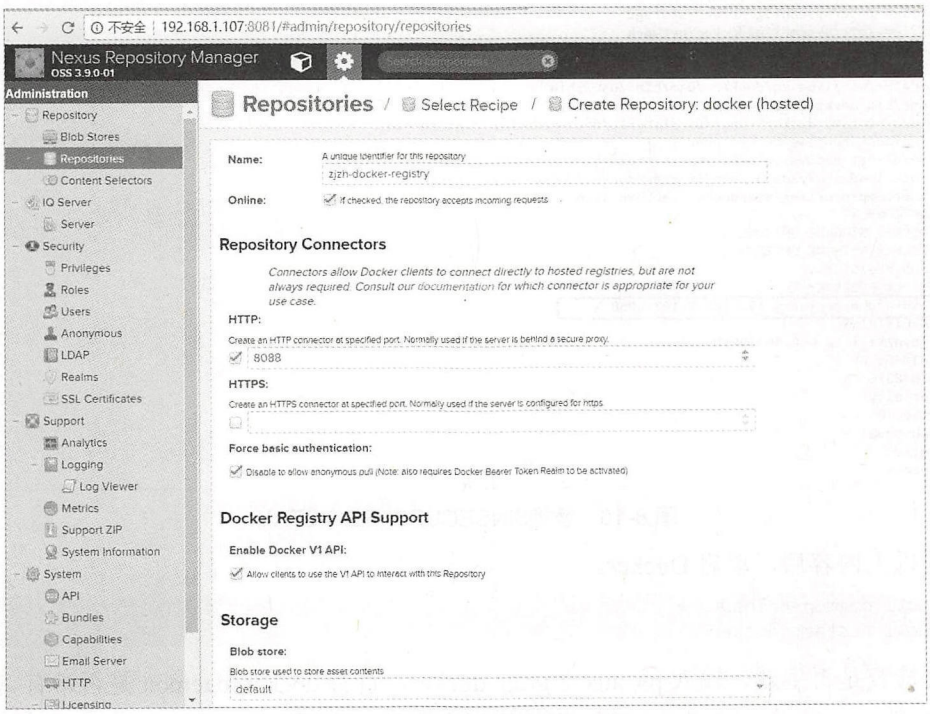


图 8-15　选择 docker（hosted）

8.3.3 配置 Docker 环境

在场景中并没有为镜像仓库服务启用 HTTPS 证书，所以 Docker 启动进程的参数还需要添加参数——inscure-registry 192.168.1.107:8088。不同的操作系统方法不一样，其中 RHEL/CentOS 7.x 的方法如下：RHEL/CentOS 7.x 使用 systemd 代替了 service，要查看 systemd 了解 Docker 服务配置。

```
systemctl status docker|grep Load
```

发现是/usr/lib/systemd/system/docker.service。

打开/usr/lib/systemd/system/docker.service，其中[Service]内容如下，

```
[Service]
Type=notify
ExecStart=/usr/bin/docker daemon -H fd://
......
```

Dockers Registry v2 默认使用证书进行认证，如要关闭认证，则需要在启动 Docker 时加入参数 insecure-registry your_registry_ip:port。

将$INSECURE_REGISTRY \替换为--insecure-registry 192.168.1.107:8088 \，如图 8-16 所示。

```
[root@renchar ~]# vi /usr/lib/systemd/system/docker.service
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.com
After=network.target rhel-push-plugin.socket registries.service
Wants=docker-storage-setup.service
Requires=docker-cleanup.timer

[Service]
Type=notify
NotifyAccess=all
EnvironmentFile=-/run/containers/registries.conf
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
Environment=GOTRACEBACK=crash
Environment=DOCKER_HTTP_HOST_COMPAT=1
Environment=PATH=/usr/libexec/docker:/usr/bin:/usr/sbin
ExecStart=/usr/bin/dockerd-current \
          --add-runtime docker-runc=/usr/libexec/docker/docker-runc-current \
          --default-runtime=docker-runc \
          --exec-opt native.cgroupdriver=systemd \
          --userland-proxy-path=/usr/libexec/docker/docker-proxy-current \
          --seccomp-profile=/etc/docker/seccomp.json \
          $OPTIONS \
          $DOCKER_STORAGE_OPTIONS \
          $DOCKER_NETWORK_OPTIONS \
          $ADD_REGISTRY \
          $BLOCK_REGISTRY \
          --insecure-registry 192.168.1.107:8088 \
          $REGISTRIES
ExecReload=/bin/kill -s HUP $MAINPID
LimitNOFILE=1048576
LimitNPROC=1048576
LimitCORE=infinity
TimeoutStartSec=0
Restart=on-abnormal
MountFlags=slave
KillMode=process
```

图 8-16 替换$INSECURE_REGISTRY

改好以上内容后，重启 Docker。

```
systemctl daemon-reload
systemctl restart docker
```

最后检查是否生效。输入 ps aux | grep docker，查看 docker daemon 是否带有--insecure-registry 参数，如没有，则表示失败，需检查以上步骤，直至 ps 时能找到--insecure-registry，如图 8-17 所示。

```
[root@renchar ~]# ps -ef |grep docker
root     11480     1  0 22:26 ?        00:00:00 /usr/bin/dockerd-current --add-runtime docker-runc=/usr/libexec/docker/docker-runc-current --default-runtime=do
cker-runc --exec-opt native.cgroupdriver=systemd --userland-proxy-path=/usr/libexec/docker/docker-proxy-current --seccomp-profile=/etc/docker/seccomp.json --se
linux-enabled --log-driver=journald --signature-verification=false --storage-driver overlay2 --insecure-registry 192.168.1.107:8088
root     11485 11480  0 22:26 ?        00:00:00 /usr/bin/docker-containerd-current -l unix:///var/run/docker/libcontainerd/docker-containerd.sock --metrics-int
erval=0 --start-timeout 2m --state-dir /var/run/docker/libcontainerd/containerd --shim docker-containerd-shim --runtime docker-runc --runtime-args --systemd-cg
roup=true
root     11607  9062  0 22:36 pts/1    00:00:00 grep --color=auto docker
[root@renchar ~]#
```

图 8-17　检查是否生效

用 docker info 命令确认生效，如图 8-18 所示。

```
[root@renchar ~]# docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 1.13.1
Storage Driver: overlay2
 Backing Filesystem: xfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: journald
Cgroup Driver: systemd
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
Swarm: inactive
Runtimes: docker-runc runc
Default Runtime: docker-runc
Init Binary: docker-init
containerd version:  (expected: aa8187dbd3b7ad67d8e5e3a15115d3eef43a7ed1)
runc version: N/A (expected: 9df8b306d01f59d3a8029be411de015b7304dd8f)
init version: N/A (expected: 949e6facb77383876aeff8a6944dde66b3089574)
Security Options:
 seccomp
  WARNING: You're not using the default seccomp profile
  Profile: /etc/docker/seccomp.json
 selinux
Kernel Version: 3.10.0-693.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
Number of Docker Hooks: 3
CPUs: 1
Total Memory: 3.702 GiB
Name: renchar
ID: 5FTK:LX6P:FURV:BCDT:XSQY:24DR:WH36:2JS5:RXDK:TE7Y:CKH7:SXMI
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
 192.168.1.107:8088
 127.0.0.0/8
Live Restore Enabled: false
Registries: docker.io (secure)
```

图 8-18　确认是否生效

用 docker login 192.168.1.107:8088 命令确认私服是否登录成功，如图 8-19 所示。

```
[root@renchar ~]# docker login 192.168.1.107:8088
Username (admin): admin
Password:
Login Succeeded
[root@renchar ~]#
```

图 8-19　确认私服是否登录成功

1. Nexus3 与 Docker 联调

1）登录私服仓库。注意在 push 上传之前必须先登录。

```
docker login 192.168.1.107:8088
```

2）打标记。在上传镜像之前需要先打一个 tag，用于版本标记。

格式如下所示。

```
docker tag <imageId or imageName> <nexus-hostname>:<repository-port>/<image>:<tag>
```

例如：

```
# docker tag 192.168.1.107:8088/demo:0.0.1-SNAPSHOT
```

3）上传镜像。

```
# docker push 192.168.1.107:8088/demo:0.0.1-SNAPSHOT
```

4）上传完成后，在 Nexus3 中查看，如图 8-20 所示。

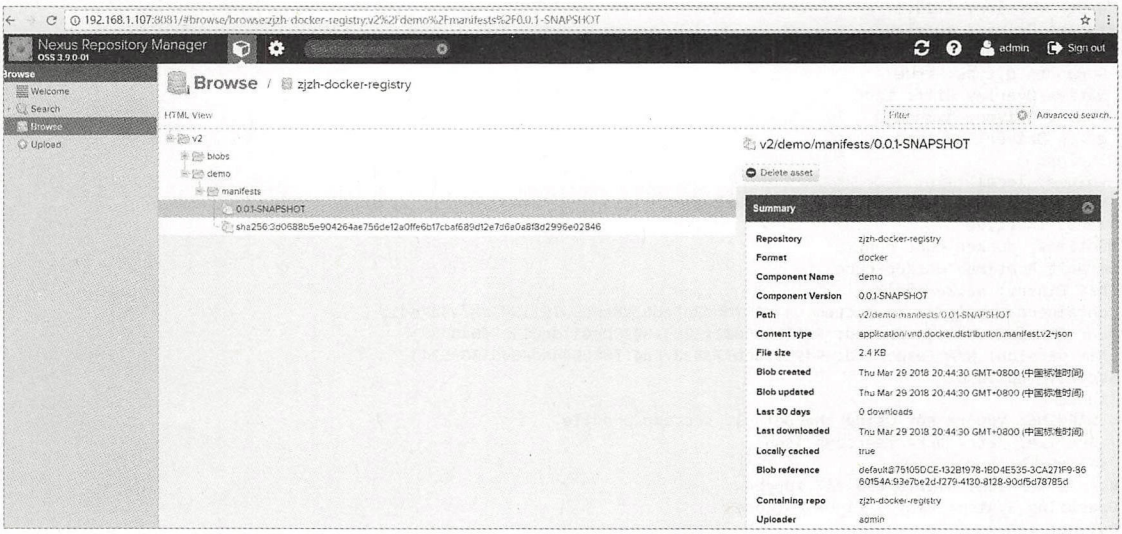


图 8-20　在 Nexus 3 中查看

5）下载镜像。从私服中下载镜像也很简单，执行以下命令即可，如图 8-21 所示。

```
# docker pull 192.168.1.107:8088/demo:0.0.1-SNAPSHOT
```

```
[root@renchar ~]# docker pull 192.168.1.107:8088/demo:0.0.1-SNAPSHOT
Trying to pull repository 192.168.1.107:8088/demo ...
0.0.1-SNAPSHOT: Pulling from 192.168.1.107:8088/demo
7448db3b31eb: Already exists
c36604fa7939: Already exists
29e8ef0e3340: Already exists
a0c934d2565d: Already exists
a360a17c9cab: Already exists
cfcc996af805: Already exists
2cf014724202: Already exists
4bc402a00dfe: Already exists
5d78ba0ea648: Pull complete
801350198b6c: Pull complete
Digest: sha256:3d0688b5e904264ae756de12a0ffe6b17cbaf689d12e7d6a0a8f8d2996e02846
Status: Downloaded newer image for 192.168.1.107:8088/demo:0.0.1-SNAPSHOT
```

图 8-21　从私服中下载镜像

6）搜索镜像。搜索镜像也与之前的类似，如图 8-22 所示。

```
# docker search 192.168.1.107:8088/demo
```

```
[root@renchar ~]# docker search 192.168.1.107:8088/demo
INDEX          NAME                                        DESCRIPTION   STARS     OFFICIAL   AUTOMATED
192.168.1.107  192.168.1.107:8088/demo:0.0.1-SNAPSHOT                    0
```

图 8-22　搜索镜像

2. Rancher 中配置 Nexus3 作为私服库

由于 Nexus3 没有配置 HTTPS 证书，因此对于 Rancher 也不安全。所以，要和原生 Dockers 环境一样先配置不安全的镜像库。

1）Rancher 中配置镜像库，如图 8-23 所示。

图 8-23　Rancher 中配置镜像库

2）输入镜像库的地址和用户信息，如图 8-24 所示。

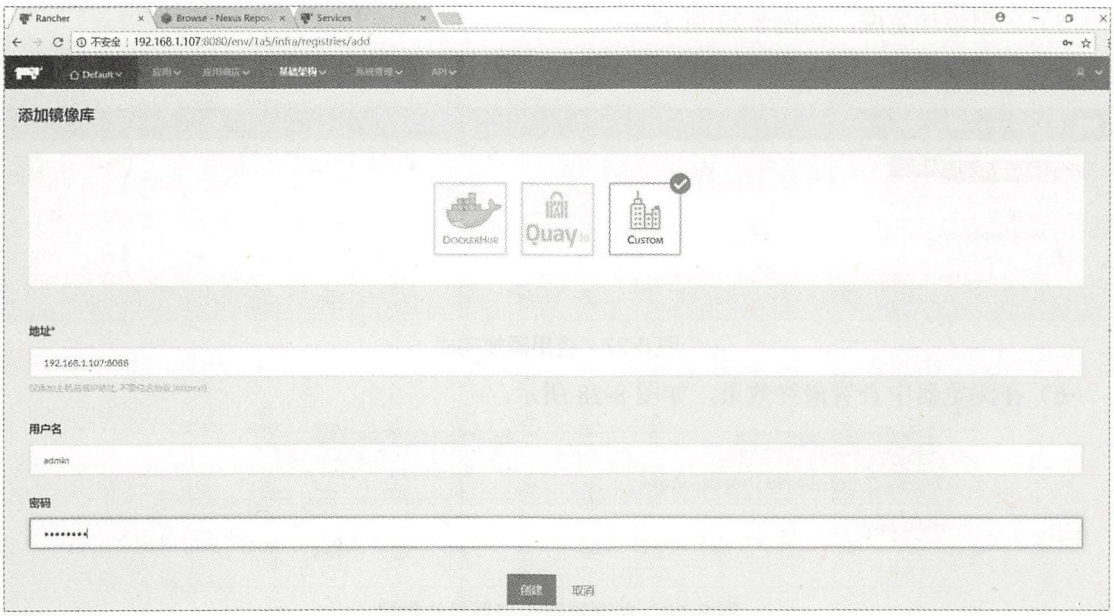


图 8-24　输入镜像库的地址和用户信息

3）创建应用，如图 8-25 所示。

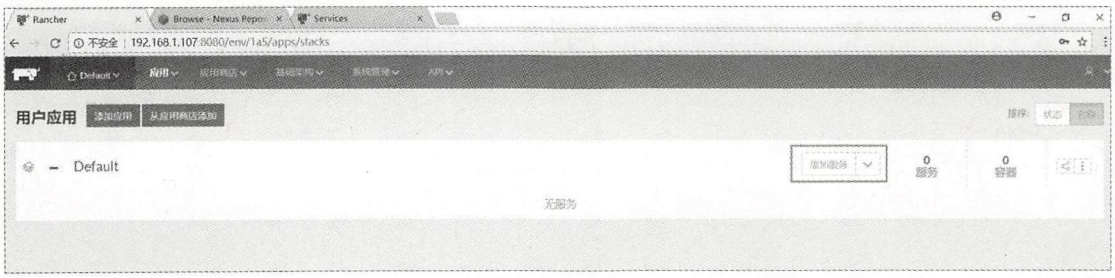


图 8-25　创建应用

4）输入镜像信息，如图 8-26 所示。

图 8-26　输入镜像信息

5）应用添加完成，如图 8-27 所示。

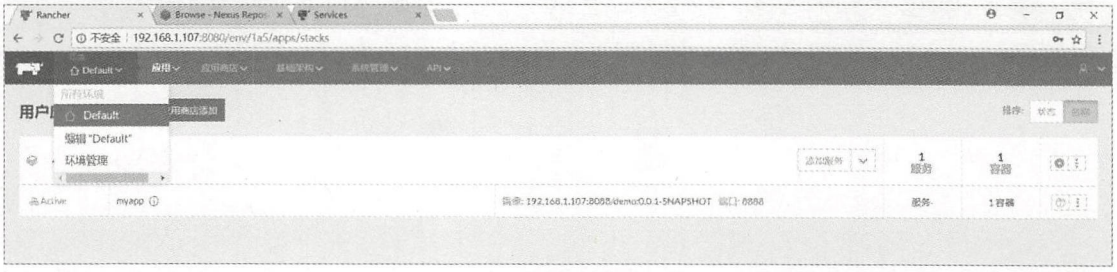


图 8-27　应用添加完成

6）在浏览器中查看最终效果，如图 8-28 所示。

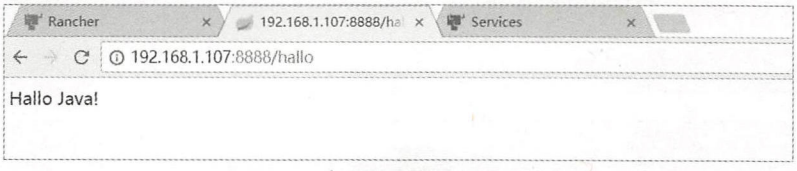


图 8-28　在浏览器中查看最终效果

7）进入容器，单击应用名称，如图 8-29 所示。

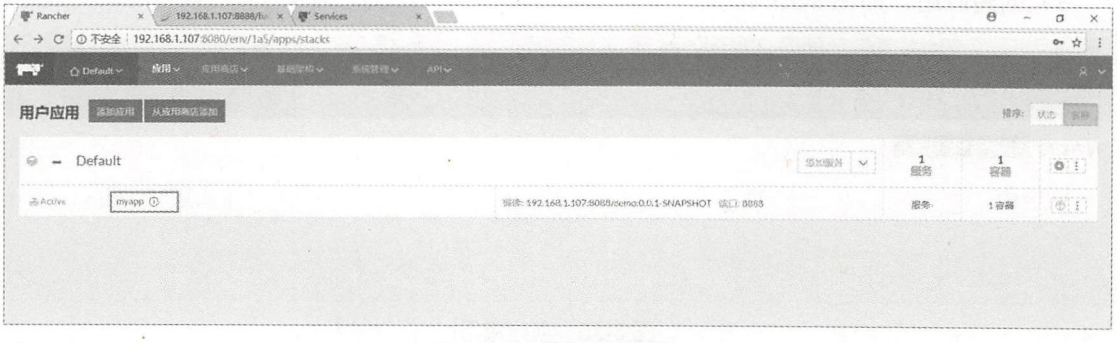


图 8-29　单击应用名称

8）执行命令行，如图 8-30 所示。

图 8-30　执行命令行

9）显示命令行窗口，如图 8-31 所示。

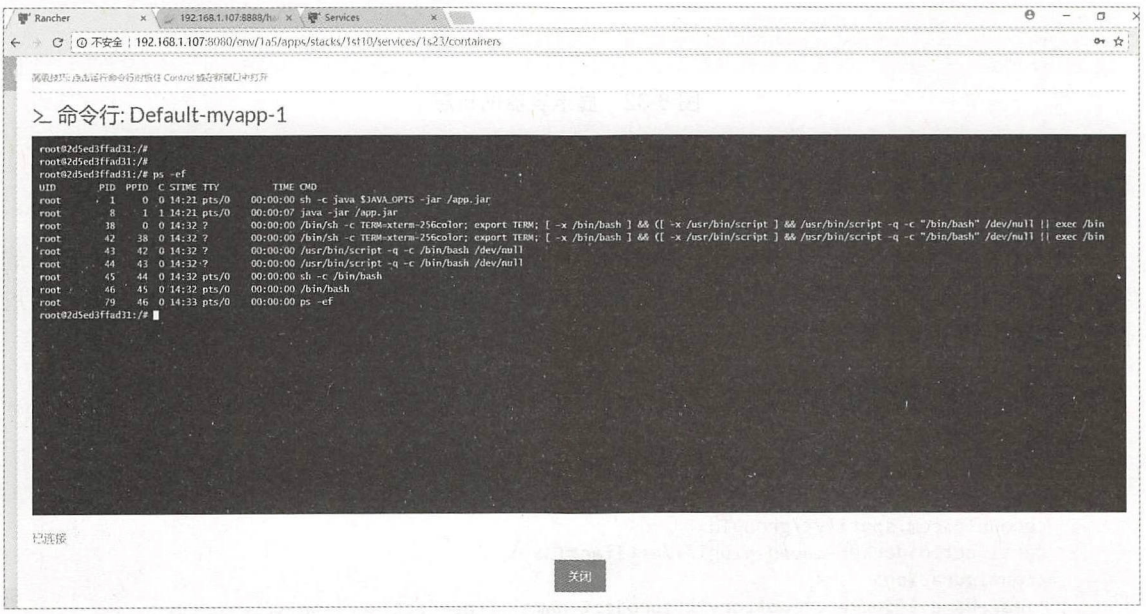


图 8-31　显示命令行窗口

10）显示容器的日志，如图 8-32 所示。

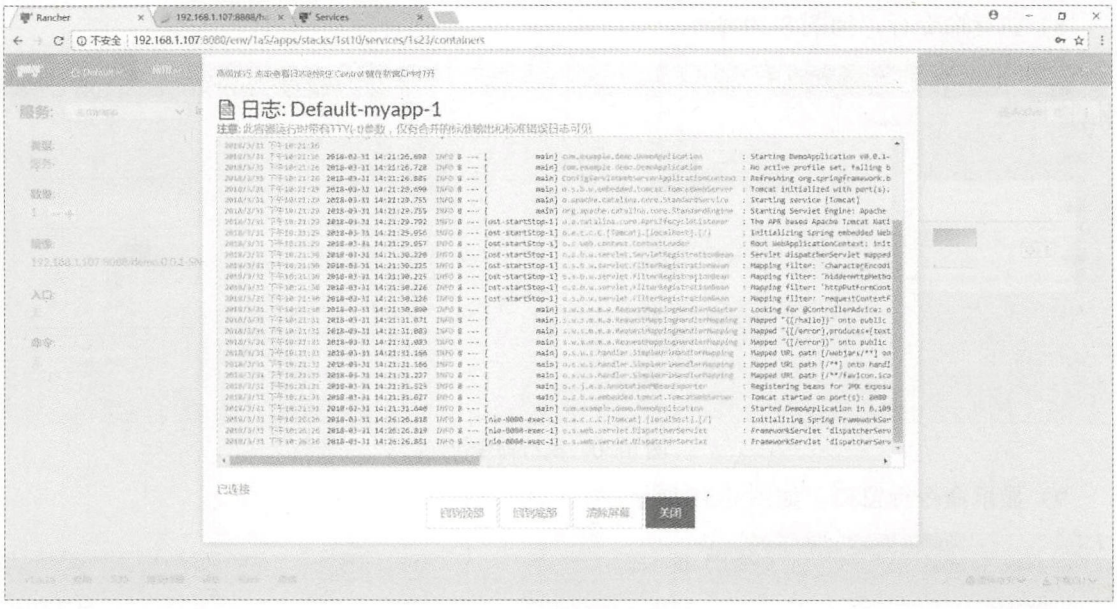


图 8-32　显示容器的日志

8.4　构建 Maven 环境

现有的持续集成功能非常多，有 Jenkins、Hunson、Bamboo 等。但这些工具有一个普遍的特点是需要配置不同的插件，且插件的种类比较多。为此，我们在 Maven 上做了改动，通过配置 Maven 的 POM 文件实现。

8.4.1　配置 POM 文件

在模块的 POM 文件中增加以下内容。

```
<plugin>
 <groupId>com.spotify</groupId>
 <artifactId>docker-maven-plugin</artifactId>
 <configuration>
 <imageName>${docker.repostory}/${project.name}:${project.version}</imageName>
  <dockerDirectory>${project.basedir}/src/main/docker</dockerDirectory>
  <serverId>docker-registry</serverId>
  <registryUrl>${docker.registryUrl}</registryUrl>
  <skipDockerBuild>false</skipDockerBuild>
  <resources>
   <resource>
    <directory>${project.build.directory}</directory>
    <include>${project.build.finalName}.jar</include>
   </resource>
  </resources>
 </configuration>
</plugin>
```

在 POM 文件的属性配置部分中增加下列内容。

```
<properties>
  <docker.repostory>192.168.1.107:8088</docker.repostory>
  <docker.registryUrl>http://192.168.1.107:8081/repository/zjzh-docker-registry/
</docker.repostory>
</properties>
```

其中，docker.repostory 的配置是 Docker 的私服配置，与 Docker 环境配置部分讲述的 --insecure-registry 参数要保持一致。

8.4.2　配置 DockerFile 文件

在各模块的 src/main 下面新建 Docker 包，在 src/main/docker 下面新建 Dockerfile 文件。这里的示例项目比较简单，是基于 Spring Boot 的一个 Web 例子，具体 DockerFile 如下。

```
FROM java
VOLUME /tmp
ADD demo-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
ENV JAVA_OPTS=""
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -jar /app.jar" ]
```

8.4.3　开启 Docker 的远程接口

修改 docker 配置文件。

```
#vi /usr/lib/systemd/system/docker.service
```

进入编辑模式后，将 ExecStart 一行后面加上如下代码。

```
-H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock
```

改完后如下所示。

```
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock
```

这里只写 0.0.0.0，不要改成自己的 IP。保存后退出，重新加载配置文件。

```
#systemctl daemon-reload
```

启动 Docker。

```
#systemctl start docker
```

输入以下代码。

```
#netstat -anp|grep 2375
```

新建 DOCKER_HOST，如图 8-33 所示。

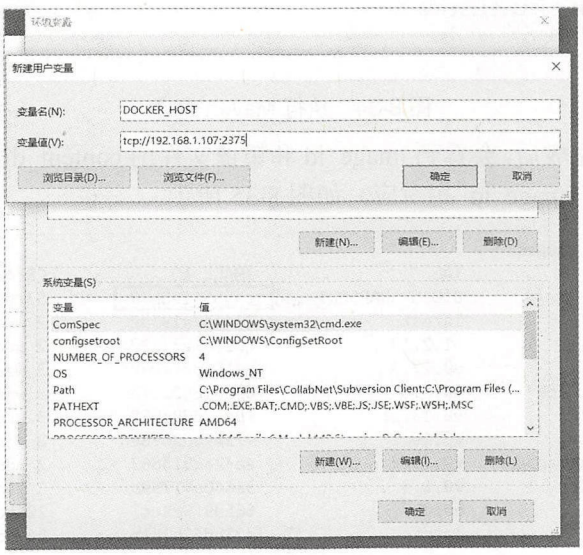


图 8-33　新建 DOCKER_HOST

显示 Docker 正在监听 2375 端口，输入以下代码。

```
#curl 127.0.0.1:2375/info
```

显示一些信息，说明远程 API 好了。

在 Windows 系统环境变量中新建 DOCKER_HOST，值为 tcp://192.168.1.107:2375，读者请改成自己的 Docker 服务器 IP 地址。

8.4.4 执行 Maven 编译

打开 CMD 命令行，进入到要编译的项目文件夹下。输入 mvn clean package docker:build -DpushImage -DskipTests，然后等待，直到最后构建成功，如图 8-34 所示。

```
 ---> a5a7f44fded9
Removing intermediate container b62dbdb9d573
Step 5/6 : ENV JAVA_OPTS ""
 ---> Running in 61fde301e6b8
 ---> 788a7f361a52
Removing intermediate container 61fde301e6b8
Step 6/6 : ENTRYPOINT sh -c java $JAVA_OPTS -jar /app.jar
 ---> Running in a7f6f3f87053
 ---> cf555f1e8b70
Removing intermediate container a7f6f3f87053
Successfully built cf555f1e8b70
[INFO] Built 192.168.1.108:8088/demo:0.0.1-SNAPSHOT
[INFO] Pushing 192.168.1.108:8088/demo:0.0.1-SNAPSHOT
The push refers to a repository [192.168.1.108:8088/demo]
e16f20f6f53a: Pushed
ee9fd40d91ee: Pushed
35c20f26d188: Layer already exists
c3fe59dd9556: Layer already exists
6ed1a81ba5b6: Layer already exists
a3483ce177ce: Layer already exists
ce6c8756685b: Layer already exists
30339f20ced0: Layer already exists
0eb22bfb707d: Layer already exists
a2ae92ffcd29: Layer already exists
0.0.1-SNAPSHOT: digest: sha256:77bf7dae92a47dbb210b962c3f804bd43a56749ac477621cdff6237460f63359 size: 2424
null: null
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12.235 s
[INFO] Finished at: 2018-04-05T11:51:31+08:00
[INFO] Final Memory: 57M/393M
[INFO] ------------------------------------------------------------------------
```

图 8-34 执行 Maven 编译

在用 Maven 编译完成后，会看到 image_id 和镜像文件的 content_digest。其中，image_id 与 docker image 命令中的 image_id 相对应，如图 8-35 所示。

```
[root@renchar ~]# docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
192.168.1.108:8088/demo             0.0.1-SNAPSHOT      cf555f1e8b70        53 seconds ago      675 MB
docker.io/rancher/server            latest              24c147a193a2        2 weeks ago         1.09 GB
docker.io/rancher/agent             v1.2.10             6023e1a77132        2 weeks ago         237 MB
docker.io/rancher/dns               v0.17.3             af151d7fa0e8        3 weeks ago         243 MB
docker.io/rancher/healthcheck       v0.3.6              db2d1e6261f3        3 weeks ago         385 MB
docker.io/rancher/net               v0.13.11            fde12dd0d058        3 weeks ago         311 MB
docker.io/rancher/metadata          v0.10.2             77299bd2078e        2 months ago        245 MB
docker.io/rancher/network-manager   v0.7.20             eaf2cdb14863        2 months ago        256 MB
docker.io/rancher/scheduler         v0.8.3              3e640a41799a        3 months ago        242 MB
docker.io/rancher/net               holder              665d9f6e8cc1        12 months ago       267 MB
docker.io/java                      latest              d23bdf5b1b1b        14 months ago       643 MB
[root@renchar ~]#
```

图 8-35 image_id 和镜像文件的 content_digest

用 docker inspect　cf555f1e8b70 命令可以看到更具体的信息，如图 8-36 所示。

```
[root@renchar ~]# docker inspect  cf555f1e8b70
[
    {
        "Id": "sha256:cf555f1e8b70cc47db32dcdd650e7008dcaa7d6b21003a996fa95d689c22588c",
        "RepoTags": [
            "192.168.1.108:8088/demo:0.0.1-SNAPSHOT"
        ],
        "RepoDigests": [
            "192.168.1.108:8088/demo@sha256:77bf7dae92a47dbb210b962c3f804bd43a56749ac477621cdff6237460f63359"
        ],
        "Parent": "sha256:788a7f361a523b4859b19c3fbbec5c917c2921742fad2680b6a763716ad131f9",
        "Comment": "",
        "Created": "2018-04-05T03:51:26.644613634Z",
        "Container": "a7f6f3f870537112c7d2af66a3bcbe4d244047823891b72fbe7be4b8a5527276",
        "ContainerConfig": {
            "Hostname": "33842653d6db",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "LANG=C.UTF-8",
                "JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64",
                "JAVA_VERSION=8u111",
                "JAVA_DEBIAN_VERSION=8u111-b14-2~bpo8+1",
                "CA_CERTIFICATES_JAVA_VERSION=20140324",
                "JAVA_OPTS="
            ],
            "Cmd": [
```

图 8-36　更具体的 content_digest 信息

与此同时，在 Nexus 3 里也可以看到镜像的相关信息，如图 8-37 所示。

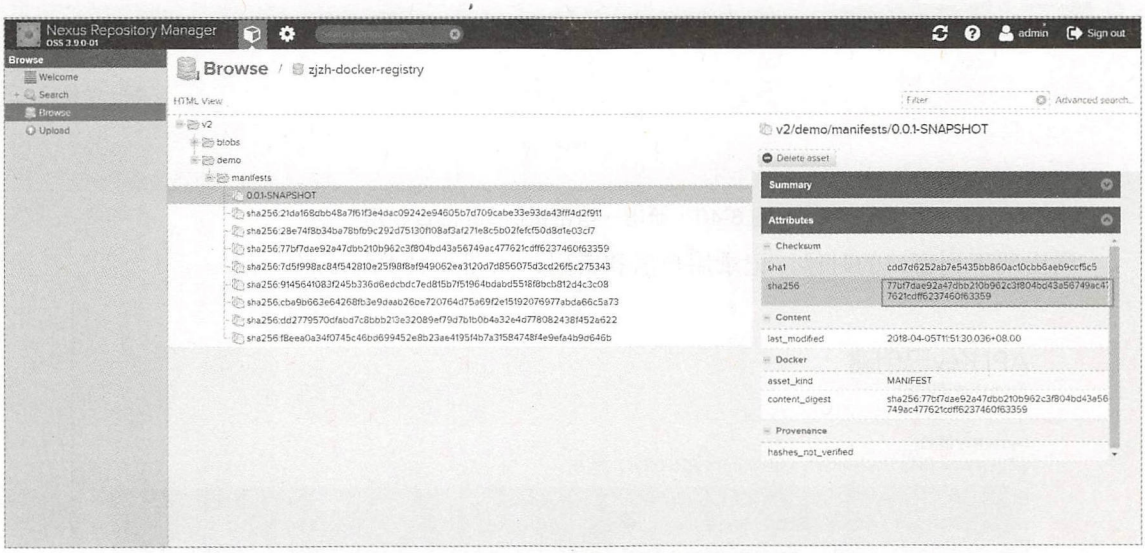


图 8-37　Nexus 3 里镜像的相关信息

8.5　Rancher 在 Jenkins 中的配置

在 Jenkins 中拉取源代码，Maven 编译的环境不再详细介绍。在这里注意的是，Jenkins 在调

用 Maven 编译后，把镜像发布到 Nexus 3 以后，通知 Rancher 进行服务升级的内容。

8.5.1 Jenkins 中安装 Rancher 插件

在 Jenkins 的插件管理部分，选择搜索 Rancher 插件，如图 8-38 所示。

图 8-38 搜索 Rancher 插件

如果搜索不到，则可以单击“立即获取”按钮，如图 8-39 所示。

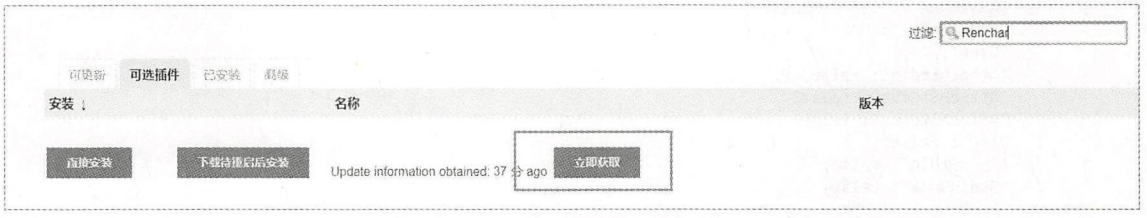


图 8-39 单击“立即获取”按钮

8.5.2 在 Rancher 服务中配置 API 连接信息

在 API 密钥的地方，新增一组密钥供 Jenkins 使用，如图 8-40 所示。

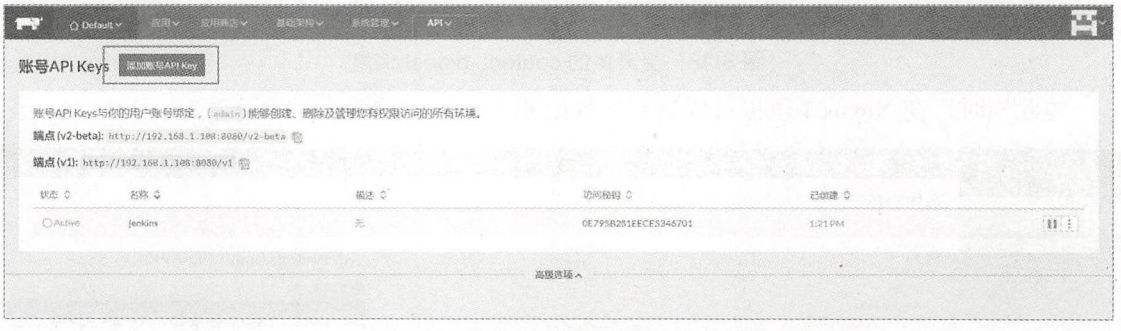


图 8-40 新增一组密钥供 Jenkins 使用

当出现下列窗口的时候，记录用户名和密码，如图 8-41 所示。

图 8-41 记录用户名和密码

8.5.3　在 Jenkins 中配置

在 Jenkins 中的构建部署环节中配置 Rancher 的一些信息，如图 8-42 所示。

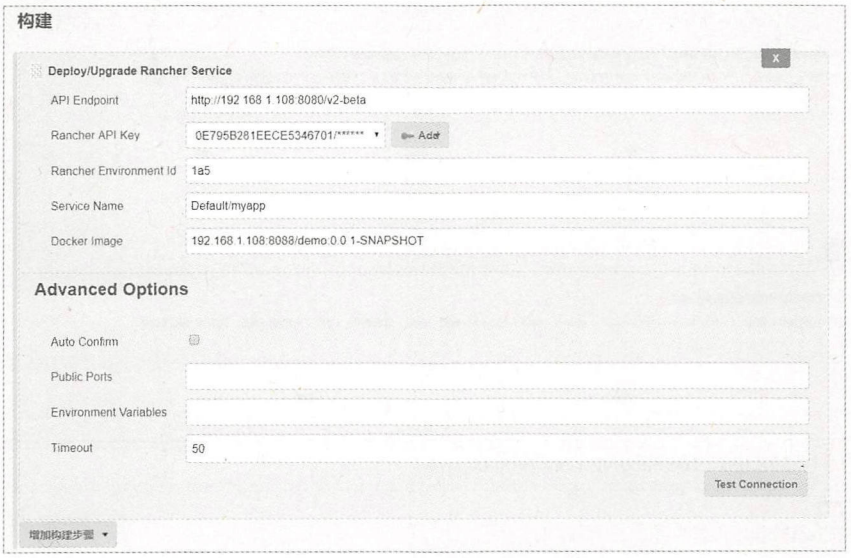


图 8-42　构建部署环节

API Endpoint 信息来源于 Rancher，如图 8-43 所示。

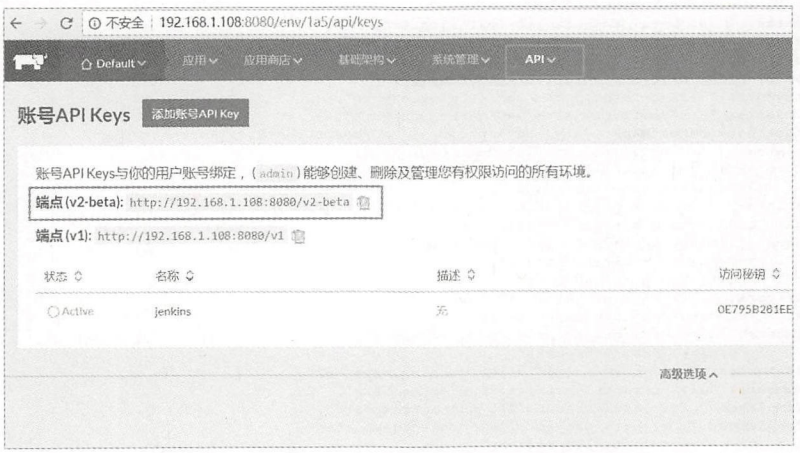


图 8-43　线上 RancherAPI 的调用端点

Rancher API Key 就是上面新增密钥时输出的一组密钥，如图 8-44 所示。

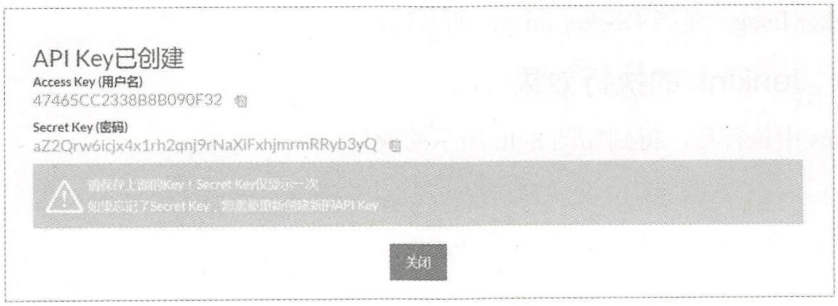


图 8-44　Rancher API Key

Rancher Environment ID 是指 Rancher 的环境的 ID 代码，在如图 8-45 所示的地方。

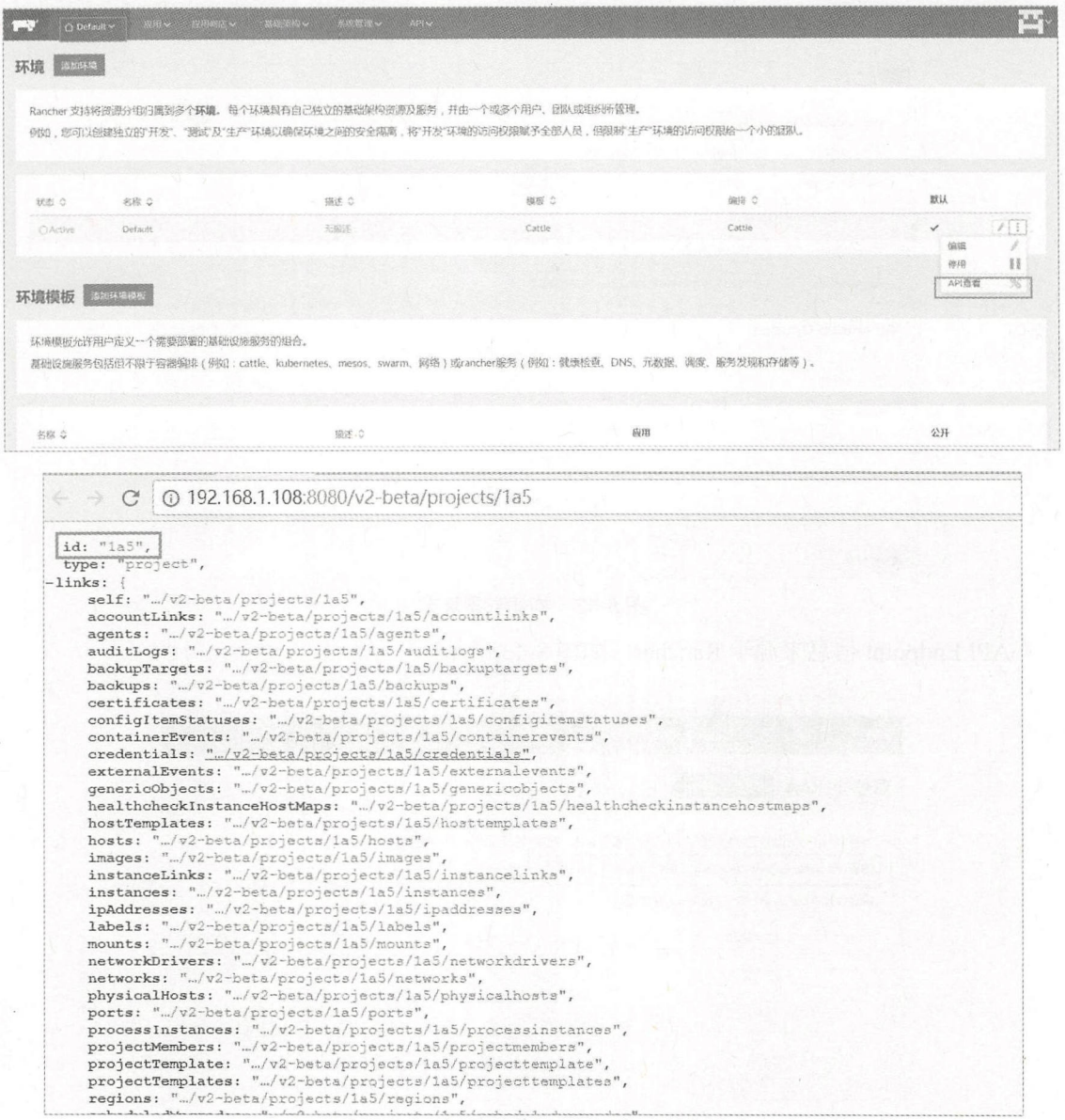


图 8-45　Rancher Environment ID

Service Name 是指服务的环境名加应用名称。本例中环境的名称是 Default，应用名称是 myapp。Docker Image 是指 Docker image 的名称。

8.5.4　Jenkins 的执行效果

在 Jenkins 中执行后，得到如图 8-46 所示的效果。

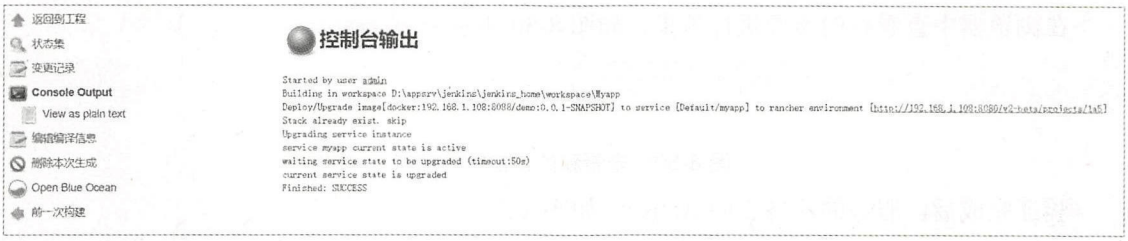


图 8-46　在 Jenkins 中执行的效果

然后在 Rancher 中会看到应用已经处于发布状态了，如图 8-47 所示。

图 8-47　应用处于发布状态

需要人工确认是否升级成功，如图 8-48 所示。

图 8-48　人工确认是否升级成功

此时进行服务验证。如果发布成功，则单击升级完成；如果验证不通过，则单击回滚，如图 8-49 所示。

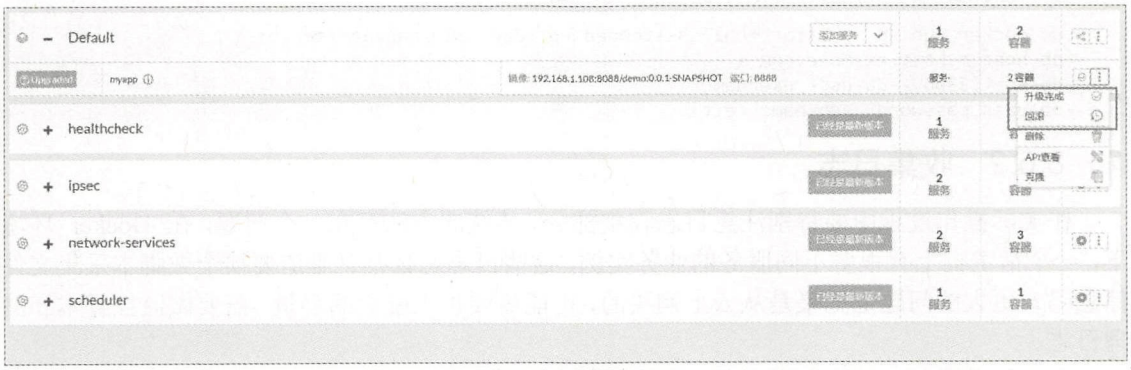


图 8-49　服务验证

在浏览器中查看新的镜像执行效果，如图 8-50 所示。

图 8-50　查看新的镜像执行效果

验证完成后，服务的状态变回 Active，如图 8-51 所示。

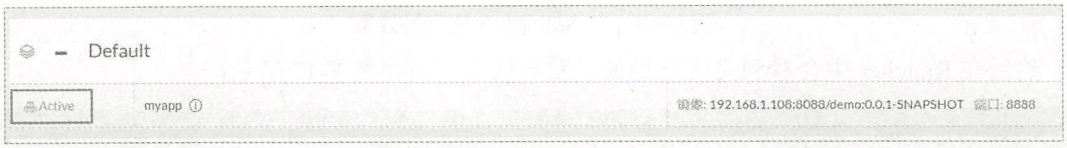


图 8-51　服务的状态为 Active

在 Active 状态下，可以进行下一次的 Jenkins 发布，否则 Jenkins 会报错，如图 8-52 所示。

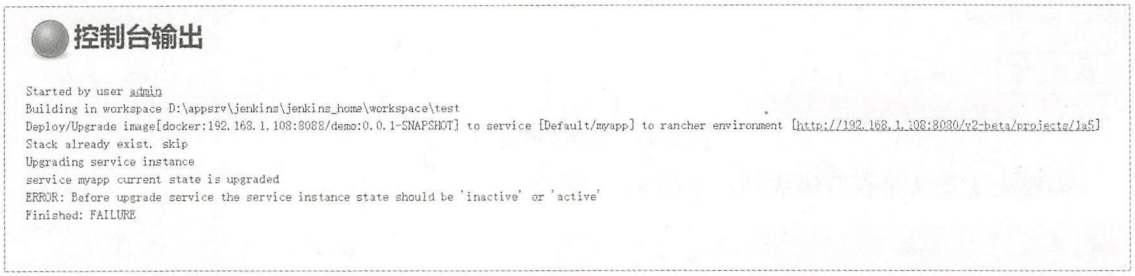


图 8-52　非 Active 状态下的 Jenkins 报错

8.6　问题与总结

8.6.1　Rancher 的高可用

在高可用（HA）的模式下运行 Rancher Server，需要暴露一个额外的端口，将额外的参数添加到启动命令中，并且运行一个外部的负载均衡。创建数据库和用户如下。

```
CREATE DATABASE IF NOT EXISTS cattle COLLATE = 'utf8_general_ci' CHARACTER SET = 'utf8';
GRANT ALL ON cattle.* TO 'cattle'@'%' IDENTIFIED BY 'cattle';
GRANT ALL ON cattle.* TO 'cattle'@'localhost' IDENTIFIED BY 'cattle';
```

在 Rancher 服务的启动命令中添加参数，如下所示。

```
sudo docker run -d --restart=unless-stopped -p 8080:8080 rancher/server \
--db-host myhost.example.com \
--db-port 3306 --db-user username \
--db-pass password --db-name cattle
```

8.6.2　收集日志

在实际操作过程中要特别注意日志收集部分。传统的方法存在一个问题，在 Docker 环境下，一个宿主机上有很多不同服务的业务实例，这些日志怎么办？难道要挨个做脚本采集文件信息吗？更大的问题是如果是从云上购买的，可能需要扩大磁盘的容量，需要做监控脚本和清理日志。

在进行微服务、容器化改造后，第一件要做的事就是让日志减少，这样就没有文件。通过对 Docker Std out 输出的日志进行采集的方式，用 Logstash 采集后，直接送到统一日志数据，中间不会产生一个日志文件，不需要清理。

例如，用 fluentd 收集日志。

```
docker run -d \
--log-driver=fluentd \
--log-opt fluentd-address=10.2.3.4:24224 \
--log-opt tag="docker.{{.Name}}" \
```

例如，用 syslogd 收集日志。

```
docker run -d \
--log-driver=syslog \
--log-opt syslog-address=10.2.3.4:514
```

8.6.3　监控告警

相对于传统的虚拟化环境，Docker 的稳定性仍然差一点。在使用了 Docker 后会经常发现某些 Docker 进程会莫名其妙地被挂起或消失。这种情况在开发测试环境中问题不大，但是在生产环境中是致命的。所以，在生产环境中应用 Docker 时，必须有一套监控告警系统与之配套，在 Docker 挂起或中断的时候要能够自动恢复并及时发出告警信息。

另外，对 IT 系统来讲，如果出现问题，则要确保在客户没有投诉之前快速地发现并处理这些问题。这时需要对收集的日志做一个日志级别的规范和定义，比如定义了 error、info 等，当一个第三方的数据源宕机时，输出一个 error 级别的日志。这个日志一定要通知自己，这就需要一个规范。我们从 ES 上会每分钟传来这些日志，会看里面 error 级别的日志对应的服务器是哪一台，并把它发送给负责的开发人员。

8.6.4　调用链监控

在一个单体里面，客户发来一个请求，数据的请求、对数据的处理、模型的计算都在一个单体应用里面完成，所有的日志都能看到是在同一个线上 ID 下面（假设非多线程情况下），会有很多日志，很容易关联。

但拆分服务后，难度显然变大了。比如 Logstash 收集一个主机采集的日志后，可以直接通过 ID 看到上下文所有打印的日志情况，进行故障、客户反馈上面的查询、分析哪个环节出了问题等。

而当拆分成很多服务之后，第一个可能面临的问题是日志收集的问题，另一个则是整个调用链的问题。针对调用链监控的问题，我们用基于 Spring Cloud 的 Seluth+Zipkin 的解决方案，如图 8-53 所示。

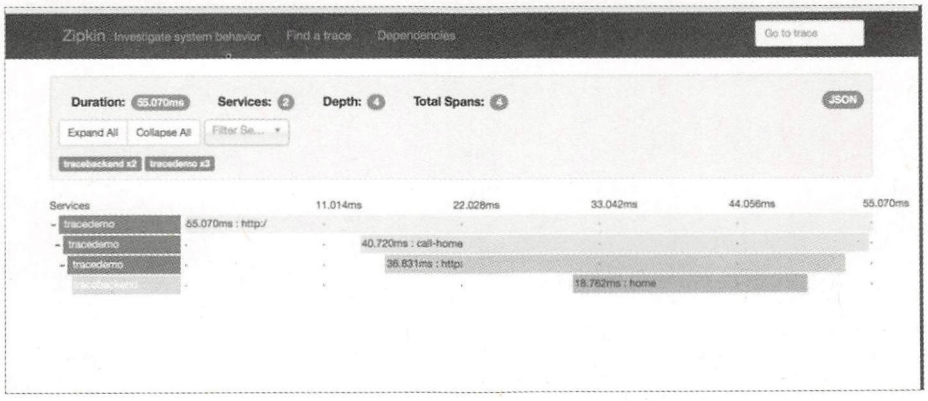


图 8-53　基于 Spring Cloud 的 Seluth+Zipkin 的解决方案

8.7 写在最后

在互联网时代，一个企业的核心竞争力就是它的服务交付能力，如何能够提升这种能力是工程师们矢志不渝的奋斗目标。通过借鉴微服务理念进行系统重构，运用 DevOps 思想进行过程改进，采用容器化技术重塑系统支撑，通过一系列的技术改造与创新，可以极大地提升企业的服务交付能力与核心竞争力。

本章作者：裘宏骏。

CHAPTER

9

第9章 九言科技Kubernetes实践

本章以九言科技遇到的问题出发，介绍引入 Kubernetes 的过程，并介绍在使用和维护过程中遇到的问题及解决方法。

9.1 现有维护中的瓶颈

随着公司业务的发展而产生的服务复杂性上升，多项目快速迭代，产生了对多个开发环境的需求。

如何快速地部署一套环境，类似淘宝的一键建站，可用于快速部署一套环境，还可用于压力测试时边压测边扩容的场景。

运维人员需要通过排查故障积累更多的知识与经验，而维护本身又希望故障越少越好，两者有根本性的矛盾。现在的智能运维也需要大量的故障处理实例进行学习，所以故障模拟环境是一种很好的方式。

容器之前的“上云”只解决了物理层出现故障时，可以缩短平均故障恢复时间（Mean Time To Repair，MTTR）的问题，大部分使用者在一个 ecs 上运行着多个不同的服务，除非严格规划到一台 ecs 只运行一个服务，所以还是没有解决服务级环境的隔离。当 ecs 出现故障时，并没有解决微服务之间的调用关联关系，而想要解决故障时服务之间的调用关联关系。除了做高可用外，负载的场景中还需要业务本身有很好的调度与容错，这在企业推行起来阻力较大，人力投入成本较高。

服务之间的调用依赖关系及请求的分布式跟踪在排查跟进问题时起到很大的作用，使用业务侵入式的方案也同样代价不小。

业务服务的微服务化带来的变化主要有三点。

- 部署单元：越来越小的粒度，加快交付效率，同时增加运维的复杂度。
- 依赖方式：从依赖库到依赖服务，增加了开发者选择的自由（语言、框架、库），提高了复用效率，同时增加了治理的复杂度。
- 架构模式：从单体应用到微服务架构，架构设计的关注点从分层转向了服务拆分，架构扁平化。

面对微服务的挑战引入容器化，但 Docker 容器本身存在稳定性和健壮性问题。当容器服务出现问题时，如何能全自动快速地发现并做调度，把影响降到最低，需要容器的调度与管理。

需要理解 12-factors 的原则，不能为了用容器而用容器。

综上所述，寻找一个能在出现故障或需要扩缩容时自动调整调用的关联服务，在尽量不侵入业务的情况下做服务的自动调度及分布式跟踪就成了关键。

9.2 容器管理平台的选择

Mesos、Kubernetes、Swarm 等容器的调度与管理方案各有特色，九言科技在实践过程中考虑到 Kubernetes 为 Google 公司开源产品、社区活跃，且为 CNCF 的项目，所以选择了 Kubernetes。

9.3 环境的搭建与 CI/CD

关于 Kubernetes 环境的搭建，已经有很成熟的文档和相关文章的介绍，这里不再做太多篇幅的介绍，只对整个 CI/CD 的环境进行分析。对于 Kubernetes 的实验环境，本节只介绍部署方法，在实际生产中还需要考虑高可用，也可以详细查看本书中关于使用 Rancher 部署的方案。

9.3.1 用 kubeadm 快速搭建 Kubernetes 环境

安装相应软件，代码如下。

```
apt-get update
apt-get install -y curl apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg|apt-key add -

cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF

apt-get update
#需要先安装 Docker，在使用官网最新版本的 Docker 可能会遇到版本不匹配的情况，建议按照官方教程操作
apt-get install -y docker.io
apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

使用 kubeadm 初始化 Kubernetes 集群。

```
kubeadm init  --pod-network-cidr=10.244.0.0/16
```

这时，kubeadm 会初始化各种环境配置，包括部署证书（private ca）和密钥，然后调用 docker pull 相应的镜像。

以下部分 log 输出。

```
Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run (as a regular user):

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  http://kubernetes.io/docs/admin/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join --token c64554.15b62d72ae89cbb3 192.168.217.135:6443
```

提示：当在 log 中看到这句输出的时候，Docker 就开始 pull 对应的镜像，比较耗时。耗时取决于用户的网速，也可以使用 registry.docker-cn.com 源进行加速。

初始化 kubectl 配置。

```
  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

部署 flannel 网络。

```
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/k8s-manifests/
kube-flannel-rbac.yml

kubectl create -f  https://raw.githubusercontent.com/coreos/flannel/master/Documentation/
kube-flannel.yml
```

验证环境，查看 Pods 的状态，node 的状态变成 ready。加入另一个节点，在另外一台机器上执行以下代码。

```
kubeadm join -token c64554.15b62d72ae89cbb3 192.168.217.135:6443
```

在 master 主机上可以看到如下内容。

```
root@c-pc:~# kubectl get nodes
NAME      STATUS    AGE       VERSION
c-pc      Ready     4h        v1.10.1
c-pc2     Ready     27s       v1.10.1
```

9.3.2 Kubernetes 环境下的 CI/CD 整体架构

如图 9-1 所示，Kubernetes 环境下的 CI/CD 整体架构，其具体流程如下：

1）开发人员向 GitLab 提交代码，代码中必须包含 Dockerfile。

2）Jenkins 的 CI 流水线自动地从 GitLab 中拉取代码。

3）交给 Maven 编译代码并打包。

4）Jenkins 把打包后的 Docker 镜像推送到 Harbor 镜像仓库。

5）Jenkins 的 CI 流水线中包括了自定义脚本，根据已准备好的 Kubernetes 的 YAML 模板，将其中的变量从 QConf 中拉取替换选项。

6）将 Jenkins 发布版本到 Kubernetes cluster 中，会用到生成的 Kubernetes YAML 配置文件和项目 Docker 镜像。

7）Kubernetes cluster 会更新 Skydns 和 Ingress/traefik 的配置，根据新部署的应用名称，在 ingress/traefik 的配置文件中增加一条路由信息。

8）Jenkins 调用外部的 DNS 服务，更换 DNS 纪录，以解决外部的访问。

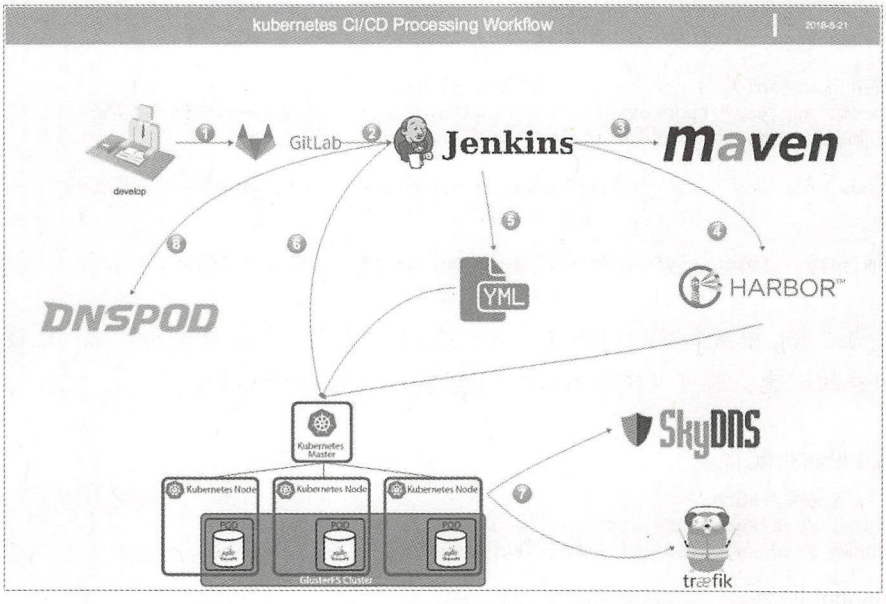


图 9-1 Kubernetes 环境下的 CI/CD 整体架构

9.4 存储引擎的选择

9.4.1 存储概述

Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含一些为运行时准备的配置参数，如匿名卷、环境变量、用户等。镜像不包含

任何动态数据，其内容在构建之后也不会被改变。

因为镜像包含操作系统完整的 Root 文件系统，其体积往往是庞大的，因此在设计 Docker 时就充分利用了 UnionFS 的技术，将其设计为分层存储的架构。

容器是在镜像基础上加一层可写容器层运行的，所以选择合适的存储引擎关系着容器运行的稳定性和性能。

9.4.2 如何选择驱动引擎

1. 根据实际项目选择存储方式

1）首先，如果使用的操作系统内核支持多个驱动，则 Docker 将按照下述优先级选择默认的存储驱动。

- 如果支持 aufs，则选择它。因为这是最早支持的驱动，尽管它并不支持所有的系统，ubuntu 系列的默认支持，redhat/centos 系列的默认不支持。
- 尽可能选择只需较少的配置即可使用的驱动，例如 Btrfs 或 ZFS（要确保后端文件系统已经配置好）。
- 最后，将尝试使用在性能和稳定性上最通用的驱动。

Overlay2 是优先级最高的，其次是 Overlay，这两个不用配置。

其次是 Device Mapper，但在生产环境使用时，要配置为 Direct-LVM 的方式，因为默认的 Loopback-LVM 尽管不用配置，但性能很差。

2）其次，要选择适合自己的业务场景的负载。

- AUFS、Overlay 和 Overlay2 文件级别的存储，更有效的利用内存，但容器的 Writable Layer 再写 I/O 压力大时增加较快，Layer 增加读写性能会下降。
- Device Mapper、Btrfs 和 ZFS 更擅长应对写 I/O 较大的业务。
- Overlay 比 Overlay2 更擅长处理小文件的写入、大量 Layers、文件系统路径很深的场景。
- Btrfs 和 ZFS 需要更多的内存。
- ZFS 是高负载的一个较好选择，例如应用到 PaaS 平台上。

2. 通过 docker info 查看存储引擎信息

核心的信息如下。

```
Server Version: 1.12.6
Storage Driver: devicemapper
 Pool Name: docker-8:2-14417928-pool
 Pool Blocksize: 65.54 kB
 Base Device Size: 10.74 GB
 Backing Filesystem: ext4
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 59.98 GB
 Data Space Total: 536.9 GB
 Data Space Available: 476.9 GB
 Metadata Space Used: 80.98 MB
 Metadata Space Total: 17.05 GB
 Metadata Space Available: 16.96 GB
 Thin Pool Minimum Free Space: 53.69 GB
Cgroup Driver: systemd
Runtimes: docker-runc runc
Kernel Version: 3.10.0-693.2.2.el7.x86_64
Operating System: CentOS Linux 7 (Core)
```

```
Total Memory: 220.2 GiB
Name: k8s-236
```

3. 使用 Device Mapper 遇到的问题

生产环境中主要的环境为 CentOS7，默认安装的 Docker 版本为 1.12，Kubernetes 为 1.6.2，存储引擎为 Device Mapper。然而，我们在使用 Device Mapper 的过程中却遇到不少问题，在 Kubernetes 环境中运行一段时间后，会出现容器卡死的情况。具体的情况如下。

主机负载很高。

```
top - 14:27:49 up 62 days, 17:45,  3 user,  load average: 218.67, 217.52, 217.23
Tasks: 1681 total,   3 running, 1676 sleeping,   0 stopped,   2 zombie
%Cpu(s):  15.9 us,  8.2 sy,  0.0 ni, 73.9 id,  0.0 wa,  0.0 hi,  2.0 si,  0.0 st
KiB Mem : 23106756+total, 67060492 free, 39881324 used, 12412576+buff/cache
KiB Swap:  8388604 total,  5920440 free,  2468164 used. 18565673+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6926 root      20   0  139988  41744   7720 R  99.7  0.0  41054:14 python
20439 root      20   0 14.747g 2.113g  17020 S  97.0  1.0  38399:41 java
 5286 root      20   0 13.867g 1.141g  16592 S  89.1  0.5  18939:00 java
40403 root      20   0    1024    564    512 S  21.1  0.0   4786:28 supervise
 2610 root      20   0 2327776 201152  51768 S  19.4  0.1   7814:19 kubelet
 2323 root      20   0 5479648 1.631g  48056 S  14.1  0.7  10523:08 dockerd
40510 root      20   0    1024    564    512 S  12.2  0.0   4359:38 supervise
40579 root      20   0    1028    564    512 R  11.8  0.0   3336:13 supervise
20621 root      20   0    1024    604    512 S   9.9  0.0   1770:29 supervise
 3262 root      20   0 2373336  37116  17528 S   8.9  0.0  28007:37 flanneld
```

经过排查分析，结果如下。

- 使用 strace 进行系统跟踪，一直处于等待状态，再无任何输出。
- 使用 perf 也没有查到明显的问题。
- 在 Jenkins 控制台的发布 Console 中，发现问题是停留在解压包的过程中（tar zxf），且解压的挂载点是本地的设备映射盘，并不是网络的挂载盘。

所以怀疑是 Docker 的存储引擎 Device Mapper 的问题。

再经过详细查找负载值的组成后发现，有大量的进程处于 D 状态，可通过 ps aux | awk '$8 ~ /^D/{print}'查看。

```
[root@k8s-236 ~]# ps aux | awk '$8 ~ /^D/{print}'
501      2809   0.0  0.0 504368 23656 ?        D    6月11   0:03 php latest/source/in/protected/yiic workersasaddscore
501      2809   0.0  0.0 504368 23656 ?        D    6月11   0:03 php latest/source/in/protected/yiic workerssolruser
501      2809   0.0  0.0 504368 23656 ?        D    6月11   0:03 php latest/source/in/protected/yiic workerssolrpaster
501      2809   0.0  0.0 504368 23656 ?        D    6月11   0:03 php latest/source/in/protected/yiic workerssolruserwatch
501      15665  0.0  0.0 473712 21240 ?        D    6月11   0:03 php-fpm: pool www
501      15666  0.0  0.0 473712 21244 ?        D    6月11   0:03 php-fpm: pool www
501      15667  0.0  0.0 473712 21248 ?        D    6月11   0:02 php-fpm: pool www
501      15668  0.0  0.0 473712 21240 ?        D    6月11   0:02 php-fpm: pool www
501      15669  0.0  0.0 473712 21232 ?        D    6月11   0:03 php-fpm: pool www
501      15670  0.0  0.0 473712 21256 ?        D    6月11   0:03 php-fpm: pool www
501      15671  0.0  0.0 473712 21244 ?        D    6月11   0:03 php-fpm: pool www
```

根据文档中 D 为 uninterruptible sleep (usually IO)不可中断的 I/O，在 Linux 的设计中对不可中断 I/O 的定义：

平均负载的概念源自 UNIX 系统，虽然各家的公式不尽相同，但都是用于衡量正在使用 CPU 的进程数量和正在等待 CPU 的进程数量，就是计算可运行或正在运行的进程的数量。所

以，平均负载可以作为 CPU 瓶颈的参考指标，如果大于 CPU 的数量，说明 CPU 可能不够用了。但是，在 Linux 系统里并不完全是这样的，Linux 系统中的平均负载除正在使用 CPU 的正在运行进程数量和正在等待 CPU 的可运行进程数量外，还包括不可中断睡眠（D 状态）的进程数量。D 状态通常发生在正在访问低速设备中，如正在从网卡固件接收网络数据包或正在从硬盘读取块数据的时候，进程会处于不可中断睡眠（D 状态）。Linux 设计时认为，不可中断睡眠应该都是很短暂的，很快就会恢复运行，所以被等同于可运行状态。然而，在现实的环境中很容易存在两种不可中断睡眠，一种是硬盘硬件发生故障时，被系统认为应该的原子读或写操作，却卡在硬件操作上；另一种是常见的网络挂载文件系统，当网络文件系统响应很慢或中断时，对于系统来说一样是按标准文件系统的原子读写操作，也很容易卡在不可中断睡眠状态上。

4. 更换存储引擎

由于Linux系统默认不支持AUFS，要让Kernel支持AUFS，得把内核更换成Kernel-lt-AUFS，也可以更换为Overlay2，而要使用Overlay2，建议更新到Kernel 4.0以上，也可以升级到Kernel-lt上。

注意： Overlay2 在 XFS 的文件系统上，需要打开 ftype。

升级版本：

1）升级主机的 Kernel 版本。将 Docker 升级到更高的版本，因为 Docker1.13 以后的版本才支持在不关闭已运行的 Docker 容器的情况下重启 Docker 服务端。

Kubernetes 支持的版本与 Docker 的版本有严格的支持范围，直接升级 Docker 本身会导致整个集群不可用。

Kubernetes 1.9 <-> Docker 1.11.2, 1.13.1, 17.03.x。

Kubernetes 1.8 <-> Docker 1.11.2, 1.13.1, 17.03.x。

Kubernetes 1.7 <-> Docker 1.10.3, 1.11.2, 1.12.6。

Kubernetes 1.6 <-> Docker 1.10.3, 1.11.2, 1.12.6。

Kubernetes 1.5 <-> Docker 1.10.3, 1.11.2, 1.12.3。

所以分析整理循环升级顺序如下：标记本 node 点标记为 kubectl cordon；再驱赶本 node 点的服务 kubectl drain；关闭 Kubernetes 及 Docker 服务；安装 kernel-lt/kernel-lt-aufs，升级 Docker 版本和 Kubernetes 版本。

添加源 https://download.docker.com/linux/centos/docker-ce.repo 和 https://yum.spaceduck.org/kernel-lt-aufs/kernel-lt-aufs.repo，然后对 yum 进行安装和升级就可以。

2）重启机器。新版本大部分的启动参数和老版本是一样的，少部分不同，报错时根据报错提示修改配置文件即可。修改 Docker 的存储引擎和 Cgroup 引擎，修改 Kubelet 启动参数 --fail-swap-on=false（新版本会报错，可以加上启动参数）。

启动服务，正常以后就可以 kubectl uncordon 了。验证升级，代码如下：

```
[root@k8s-24 yum.repos.d]# kubectl get nodes
NAME           STATUS    ROLES     AGE       VERSION
10.10.106.126   Ready     <none>    209d      v1.6.4
10.10.106.130   Ready     <none>    82d       v1.9.6
10.10.106.236   Ready     <none>    208d      v1.6.4
10.10.106.237   Ready     <none>    230d      v1.9.6
```

```
10.10.106.238   Ready     <none>    166d      v1.6.4
10.10.106.24    Ready     <none>    209d      v1.9.6
```

通过 docker info 再次查看，代码如下。

至此升级完成。

```
[root@k8s-24 ~]# docker info
Server Version: 18.03.0-ce
Storage Driver: Overlay2
 Backing Filesystem: extfs
        Supports d_type: true
        Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
        Network: bridge host macvlan null overlay
        Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: cfd04396dc68220d1cecbe686a6cc3aa5ce3667c
runc version: 4fc53a81fb7c994640722ac585fa9ca548971871
init version: 949e6fa
Kernel Version: 4.14.33-1.el7.centos.x86_64
Operating System: CentOS Linux 7 (Core)
Name: k8s-24
```

9.5 Kubernetes 日志收集

应用的日志可用于排查线上问题、记录访问信息、进一步统计和分析，在没有使用 Kubernetes 之前，公司的日志处理如下。

9.5.1 收集日志的需求

严重错误日志监控。

分析及统计型监控。

9.5.2 收集日志的解决方案

1. 服务的日志写入磁盘

使用本机上的 Zabbix 客户端进行错误日志的匹配监控。

1）优点

- 简单，不会丢失严重错误型日志的报警。
- 现有的监控就可以满足。
- 同机同一服务的报警可以做一定的收敛。

2）缺点

- 多机同服务的报警难以收敛，难以在以服务为单位上进行监控。
- 分析排查问题时，请求分布在不同机器上，排查困难。
- 无法满足统计型监控。
- 单服务日志的写入虽然是顺序追加写，但同机上同一磁盘部署多个服务，这时从整块硬盘来说，会由顺序写转入随机写，对 I/O 的影响会日益严重。

2. 日志的收集

鉴于原有 Zabbix 的监控问题和分析、统计型监控的需求，引入日志的收集。主要在两个方案中做选择。

- 服务还是把日志写入磁盘，由外部程序完成收集入库，类似于 Flume NG+Kafka+ELK 的方案。
- 服务直接把日志写入远端，如 Elasticsearch、Kafka、MongoDB。

日志写入远端需要一定程度侵入服务程序，对于有采用类似 log4j 日志框架的 Java 服务尚好，但对于 C 程序、Node.js、PHP 的程序侵入式较大。且各日志服务组件采用的方式不一，当远端的日志存储端响应慢或有问题时，可能会影响服务的正常运行。常见的情况有日志存储端异常时，服务系统大量写入日志报错，频繁操作 I/O，业务性能下降；日志存储端异常时，大量的日志对象占满内存，导致 oom；日志存储端异常时，耗尽连接池，堵塞或影响服务请求。

最后选择了 Flume NG+Kafka+ELK 方案，架构如图 9-2 所示。

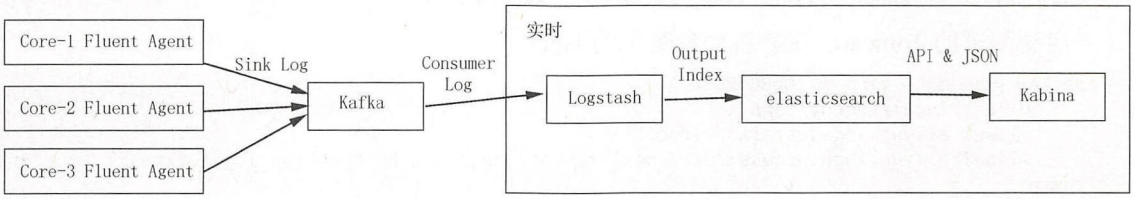


图 9-2 Flume NG+Kafka+ELK 方案架构

3. 在进入 Kubernetes 集群后又有了新的问题

1）无论使用 Zabbix 还是 Flume NG+Kafka+ELK 的方案，都会面临一个问题，原有的方案都需要事先知道要收集或监控的文件在哪，而 Kubernetes 上的 Pod 在集群中经常迁移或弹性伸缩，导致原有的方案无法感知 Kubernetes 的调度而需要频繁地修改或添加配置，直接一个 Pod 绑定一个收集程序又太重了。

为解决遇到的问题，也经过了几个阶段的发展：第一阶段，直接把日志写入网络共享盘 NFS；第二阶段，为解决 NFS 的单点问题，把共享盘切换到 GlusterFS 上；第三阶段，找一个能自动感知集群调度变化的收集器，对于不需要频繁地修改配置的，经对比后选择了阿里云开源的 Log-pilot。

4. Log-pilot

Log-pilot 是一个开源的日志采集工具，适合直接在一台机器上运行单个进程模式。Log-pilot 有以下特点：

- 一个单独 fluentd 进程，收集机器上所有容器的日志。不需要为每个容器启动一个 fluentd 进程。
- 声明式配置。使用 label 声明要收集的日志文件的路径。
- 支持文件和 stdout。
- 支持多种后端存储：Elasticsearch、阿里云日志服务、Graylog2 等。

Log-pilot 架构如图 9-3 所示。

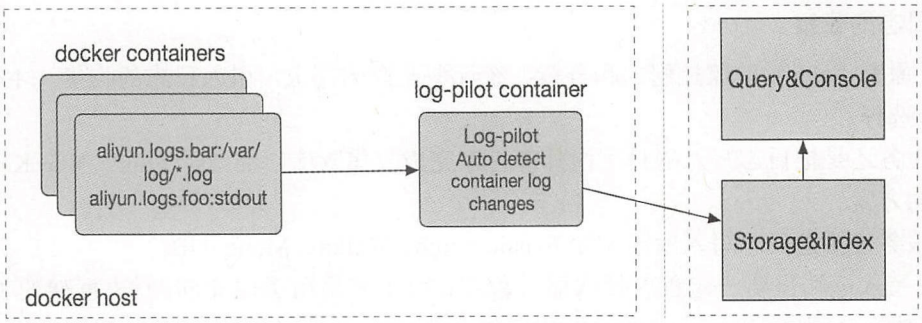


图 9-3　Log-pilot 架构

先启动好 Elasticsearch 服务，或使用已有 Elasticsearch 服务，然后启动 Log-pilot 容器。

```
docker run -d \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /:/host \
-e FLUENTD_OUTPUT=elasticsearch \
-e ELASTICSEARCH_HOST=10.10.105.131 \
-e ELASTICSEARCH_PORT=9200 \
registry.cn-hangzhou.aliyuncs.com/acs-sample/fluentd-pilot
```

启动测试的 Tomcat，注意启动参数中的 label。

```
docker run -it --rm  -p 10080:8080 \
    -v /usr/local/tomcat/logs \
    --label aliyun.logs.catalina=stdout \
    --label aliyun.logs.access=/usr/local/tomcat/logs/localhost_access_log.*.txt \
tomcat
```

这时就可以从 ELK 中看到收集的日志了，但没有标志服务，搜索时无法区分。可以在日志里添加 tag，这些 tag 相当于一些关键信息，可以附加任何需要的关联信息，这样将在搜索时更方便地把这些日志聚在一起。

1）Label 标签

- aliyun.logs.$name = $path：变量 name 是日志名称，只能包含 0-9a-zA-Z_和-变量；path 是要收集的日志路径，必须具体到文件，不能只写目录；文件名部分可以使用通配符，/var/log/he.log 和/var/log/*.log 都是正确的值，但/var/log 不能只写到目录，stdout 是一个特殊值，表示标准输出。
- aliyun.logs.$name.format：日志格式，目前支持无格式纯文本；JSON 格式，每行一个完整的 JSON 字符串；csv 格式。
- aliyun.logs.$name.tags：上报日志时额外增加的字段，格式为 k1=v1，k2=v2，每个 Key-Value 之间使用逗号分隔，例如 aliyun.logs.access.tags="name=hello,stage=test"，上报到存储的日志里会出现 name 字段和 stage 字段。

如果使用 Elasticsearch 作为日志存储，target 这个 tag 具有特殊含义，表示 Elasticsearch 里对应的 index。

这里介绍 Docker 本身支持的 logdriver，如表 9-1 所示。

表 9-1　Docker 本身支持的 logdriver

Driver	描　述
none	丢弃容器输出，docker logs 命令也看不到任何内容
json-file	默认驱动，使用 JSON 文件保存日志
syslog	日志写到 syslog 里
journald	日志发送给 journald(systemd)

续表

Driver	描　述
gelf	以 gelf 格式发送给如 Graylog 或 Logstash
fluentd	日志发送给 fluentd
awslogs	日志发送给 Amazon CloudWatch Logs
splunk	日志发送给 splunk
etwlogs	日志发送给 Event Tracing for Windows.仅支持 Windows 平台
gcplogs	日志发送给 Google Cloud Platform（GCP）Logging
logentries	日志发送给 Rapid7 Logentries

9.6 未来探索

9.6.1 Service Mesh 介绍

看到大公司的业务服务交互图监控和调用跟踪链后，有些公司也想试试。然而，为了完成这个目标，往往需要中间件团队做 SDK 让业务方集成，或让业务方做代码的侵入，推动起相当困难，进度也慢，所以都期望能用无侵入的方法完成，Service Mesh 应运而生。当然，Service Mesh 远不止于用服务交互监控和调用链功能，还有其他很强大的功能。

Service Mesh 是在服务和网络间透明地注入一层，该层将赋予操作人员对所需功能的控制，同时将开发人员从编码实现分布式系统问题中解放出来，可以将它比作是应用程序或微服务间的 TCP/IP，负责服务之间的网络调用、限流、熔断和监控。对于编写应用程序来说，一般无须关心 TCP/IP 层（比如通过 HTTP 协议的 RESTful 应用），同样使用 Service Mesh 也就无须关心服务之间的通过应用程序或其他框架实现的事情，比如 Spring Cloud、OSS，现在只要交给 Service Mesh 就可以。

Service Mesh 有如下几个特点：

- 应用程序间通信的中间层。
- 轻量级网络代理。
- 应用程序无感知。
- 解耦应用程序的重试或超时、监控、追踪和服务发现。

目前比较流行的 Service Mesh 开源软件 Istio 和 Linkerd 都可以直接在 Kubernetes 中集成，其中 Linkerd 已经成为 CNCF 成员。而 Istio 是由 Google、IBM 和 Lyft 开源的微服务管理、保护和监控框架，关注度也较高。Istio 在本书的其他章节中已有介绍，这里就不再赘述。Linkerd 已经有不少公司已经在商用，较为稳定，但对资源要求较高。

Service Mesh 主要由数据层和控制层组成，常见的如下：

数据层：Linkerd、Nginx、HAProxy、Envoy、Traefik。

数据层：Istio、Nelson、SmartStack。

在 Google group 上关于三种类型的对比和讨论，如表 9-2 所示。

表 9-2　Linkerd、Istio/ Envoy 和 Nginx 对比

名　字	Linkerd	Istio/Envoy	Nginx Plus
官网	https://linkerd.io/	https://istio.io	https://www.nginx.com/products/
许可	AL2.0	AL2.0	商业

续表

名　字	Linkerd	Istio/Envoy	Nginx Plus
维护人	Buoyant（Twitter pedigree）Buoyant	Lyft、Google、IBM	Nginx Inc
开发语言	Scala/Rust	Go/C++11	C
部署平台	Any	Kubernetes	Any
支持协议	HTTP/2，gRPC，TCP	HTTP/2，gRPC，TCP	HTTP/1.1，TCP，UDP
服务发现方式	File，Consul，ZK，etcd，k8s，Marathon	No（由外部完成）	File，Consul，ZK，etcd，k8s，Marathon（nixy）
部署方式	Per Host/Sidecar	Sidecar	Per Host/Sidecar
请求跟踪	Yes（Zipkin）	Yes（Zipkin）	Possible，But bespoke
插件语言	Java plugins	Golang plugins	C modules
商业支持	Yes（Buoyant）	No	Yes（Nginx Inc）
生产环境部署	PayPal，Expedia，AOL，Monzo	Lyft（Envoy）	WIX，ARM，BlueStem

后来 Linkerd 的厂家 Buoyant 又针对 K8S 重新开发了 Conduit，它可以透明地管理服务运行时之间的通信，使得在 Kubernetes 上运行服务更加安全和可靠；它还具有不用修改任何应用程序代码即可改进应用程序的可观测性、可靠性及安全性等方面的特性。

Conduit 与 Linkerd 的设计方式不同，它跟 Istio 一样使用的是 Sidecar 模式，但架构又没有 Istio 那么复杂。Conduit 目前只支持 Kubernetes，且只支持 HTTP2（包括 gRPC）协议。

2018 年 7 月 6 日，Linkderd 表示 Conduit 将作为 Linkerd2.0 的基础，两者合并。

Nginmesh 是 Nginx 的 Service Mesh 开源项目，用于 Istio 服务网格平台中的数据面板代理。它旨在提供七层负载均衡和服务路由功能，与 Istio 集成作为 Sidecar 部署，并将以“标准、可靠和安全的方式”使得服务间通信更容易。Nginmesh 已经有 0.7 版本，提供了服务发现、请求转发、路由规则、性能指标收集等功能，其他各项功能还不是很完善，可以关注其发展趋势，不建议使用到生产环境。

但 Nginmesh 目前也有限制：

- 不支持 TCP 和 gRPC。
- 不支持配额。
- 只能在 Kubernetes 中使用。

目前 Service Mesh 还在快速发展中，大多还处在测试和理念阶段，生产中全面使用的还较少，一方面 Service Mesh 的性能损失还是不少，另一方面现在的生产中全面用 HTTP2 和 gRPC 的覆盖还太少。目前很多产品处于全站用 HTTPS 的阶段，因为各大主流浏览器对 HTTP2 的实现是基于 HTTPS 的。

9.6.2 FaaS 与 Serverless

图 9-4 所示为云计算的分层概括，其中 Serverless 是构建在虚拟机和容器之上的一层，与应用本身的关系更加密切。

它通常包含了 BaaS（Backend as a Service）和 FaaS（Function as a Service）两个领域。

BaaS 指后端即服务，一般是一个个的 API 调用后端或别人已经实现好的程序逻辑，比如身份验证服务 OAuth、亚马逊的 RDS 可以替代自己部署的 MySQL。

FaaS 指函数即服务，FaaS 是无服务器计算的一种形式，当前使用最广泛的是 AWS 的

Lambada。FaaS 在本质上是一种事件驱动的由消息触发的服务，FaaS 供应商一般会集成各种同步和异步的事件源，通过订阅这些事件源，可以突发或定期地触发函数运行。

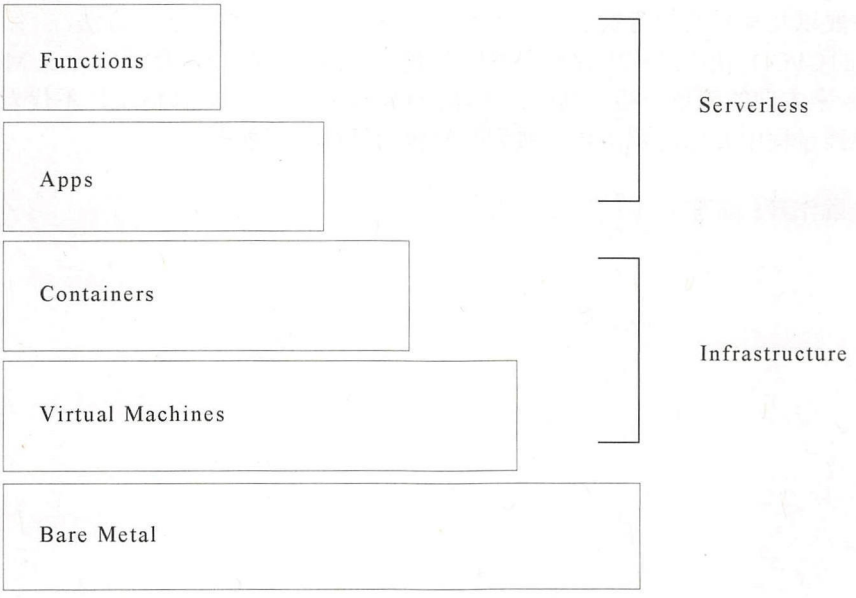


图 9-4 云计算的分层概括

OpenFaaS 是一款高人气的开源 FaaS 框架，可以直接在 Kubernetes 上运行，也可以基于 Swarm 或容器运行。

在 Kubernetes 上部署 OpenFaaS 十分简单，用到的镜像如下：

- functions/faas-netesd:0.3.4。
- functions/gateway:0.6.14。
- functions/prometheus:latest-k8s。
- functions/alertmanager:latest-k8s。

这些镜像都存储在 DockerHub 上。OpenFaaS 的架构如图 9-5 所示。

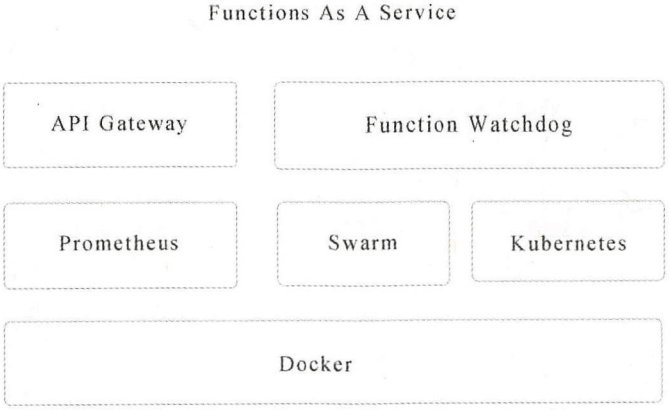


图 9-5 OpenFaaS 的架构

9.7 小结

本章主要以九言科技的背景出发，记录和分享了遇到的问题及解决方法，介绍了目前基于Kubernetes的CI/CD，使用过程中存储引擎的问题，日志收集问题，以及对Service Mesh和FaaS、Serverless未来方向的简单介绍。其中的方案选择结合了公司的具体情况，不代表开源项目的好坏，在实践过程中要以能解决目前遇到的问题为目标，不能盲从。

本章作者：苏茶林。

CHAPTER

10

第10章 沃趣科技的容器化RDS之路

“你不是不够好，你只是过时了。”这句话用在IT行业特别合适，每隔一段时间都会有新的技术出现，让程序员们应接不暇。沃趣科技（以下简称“沃趣”）数据库产品技术研发已悄然走过六个年头，技术也在发生翻天覆地的变化。回顾沃趣在数据库领域（主要以Oracle、MySQL为主）几个时期不同的技术成果及历程，结合用户对于数据库运维自动化的要求越来越高，数据库即服务（DBaaS or RDS）的需求越来越大，与大家分享沃趣在对容器化技术的引入及应用后，对下一代数据库运维架构的理解和目前正在做的工作。

10.1 容器化 RDS：计算存储分离架构下的“Split-Brain”

无论架构选型还是生活，大多数时候都是在做 trade off，享受计算存储分离带来的益处，也意味着要忍受它带来的一些棘手问题。本文尝试结合 Kubernetes、Docker、MySQL 和计算存储分离架构，分享遇到的诸多问题之一“Split-Brain”。

当前业界的数据库技术发展趋势是：数据库容器化作为下一代数据库基础架构，基于编排架构管理容器化数据库，采用计算存储分离架构。

这和沃趣在私有 RDS 上的技术选型不谋而合，尤其是计算存储分离架构，如图 10-1 所示。

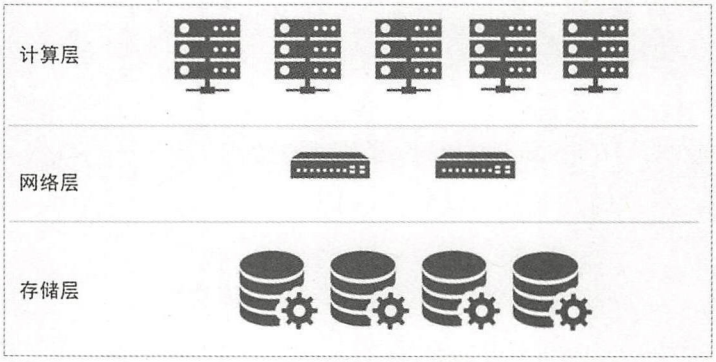


图 10-1 计算存储分离架构

以阿里巴巴为例，考虑到今时今日的规模，如果能够实现数据库服务的离线（ODPS）或在线集群的混合部署，意义极其重大。关键问题在于，离线计算和在线计算对实时性要求不同，硬件配置也不同，尤其是本地存储介质：离线以机械磁盘为主，在线以 SSD / Flash 为主。

如果采用本地存储作为数据库实例的存储介质，试想一下，一个 Storage Qos 要求使 Flash 的数据库实例无法调度到离线计算集群，哪怕离线计算集群 CPU、Memory 有大量空闲。

计算存储分离为实现离线集群或在线集群的混合部署提供了可能。

结合 Kubernetes、Docker 和 MySQL，进一步细化架构图，如图 10-2 所示。

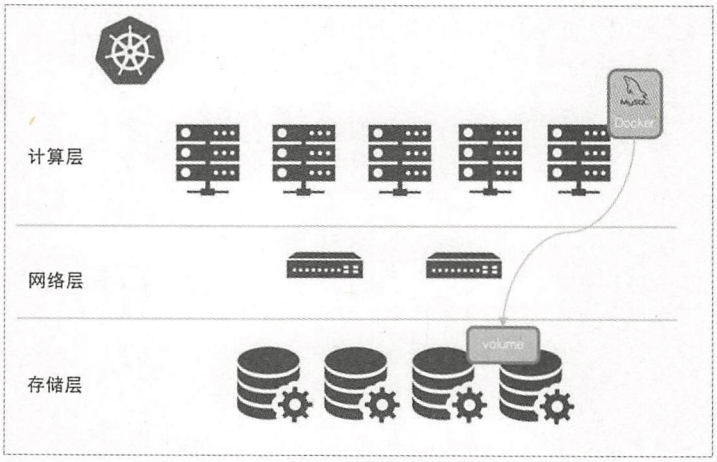


图 10-2 细化后的架构

同时，这套架构也是更加简单、通用、高效的 High Availability 方案。当集群中某个 Node 不可用时，借助 Kubernetes 的原生组件 Node Controller、Scheduler 和原生 API StatefulSet 即可

将数据库实例调度到其他可用节点，以实现数据库实例的高可用，如图 10-3 所示。

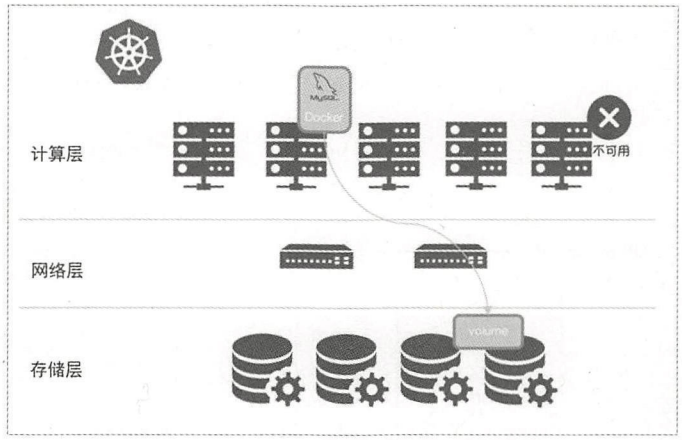


图 10-3　数据库实例的高可用方案

一切是多么的美好，是不是可以得到这个结论：借助 Kubernetes 的原生组件 Node Controller、Scheduler 和原生 API StatefulSet，加上计算存储分离架构，并将成熟的分布式文件系统集成到 Kubernetes 存储系统，就能提供私有 RDS 服务。

但是遇到了“Split-Brain”问题，也正是本文的主题。回到 High Availability 方案。判定 Node 不可用将是后续触发 Failover 动作的关键，这里需要对节点状态的判定机制稍做展开：

- Kubelet 借助 API Server 定期（node-status-update-frequency）更新 etcd 中对应节点的心跳信息。
- Controller Manager 中的 Node Controller 组件定期（node-monitor-period）轮询 etcd 中节点的心跳信息。
- 如果在周期（node-monitor-grace-period）内心跳更新丢失，该节点标记为 Unknown（ConditionUnknown）。
- 如果在周期（pod-eviction-timeout）内心跳更新持续丢失，Node Controller 将会触发集群层面的驱逐机制。
- Scheduler 将 Unknown 节点上的所有数据库实例调度到其他健康（Ready）节点。

访问架构如图 10-4 所示。

补充一句，借助 etcd 集群的高可用强一致，得以保证 Kubernetes 集群元信息的一致性：

- etcd 基于 Raft 算法实现。
- Raft 算法是一种基于消息传递（State Machine Replicated）且具有高度容错（Fault Tolerance）特性的一致性算法（Consensus algorithm）。
- Raft 是 Paxos 的简化版本。
- 如果对于 Raft 算法的实现感兴趣，可以看 https://github.com/goraft/raft。

所有对一致性算法感兴趣的同学，都值得花精力学习。基于 goraft/Raft，沃趣实现了 Network Partition Failures/Recovery TestCase。

但看上去合理的机制会带来两个问题。

问题一：无法判定节点真实状态。心跳更新是判断节点是否可用的依据，但心跳更新丢失是无法判定节点真实状态的（Kubernetes 中将节点标记为 Condition Unknown 也说明了这点）。Node 可能仅仅是网络问题，CPU 繁忙、“假死”、Kubelet Bug 等原因导致心跳更新丢失，但节点上的数据库实例还在运行中。

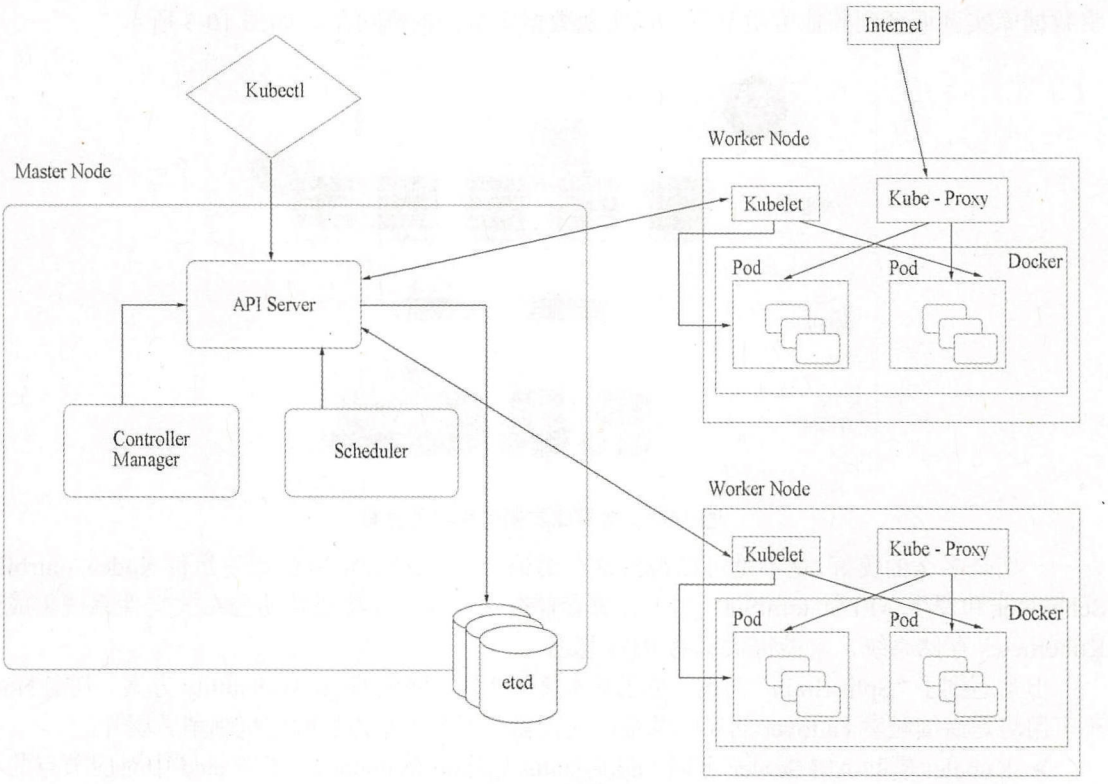


图 10-4　访问架构

问题二：缺乏有效的 Fence 机制。在这种情况下，借助 Kubernetes 的原生组件 Node Controller、Scheduler 和原生 API StatefulSet 实现的 Failover，将数据库实例从 Unknown 节点驱逐到可用节点，但对原 Unknown 节点不做任何操作。

这种“软驱逐”将会导致新旧两个数据库实例同时访问同一份数据文件，如图 10-5 所示。

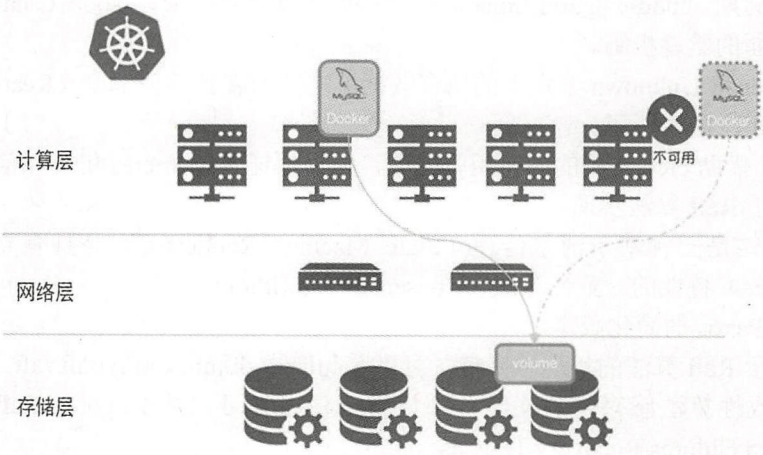


图 10-5　“软驱逐”效果示意图

发生“Split-Brain”导致 Data Corruption 数据丢失，损失将无法弥补。

下面是故障复现，通过日志和代码分析驱逐的工作机制，总结“Split-Brain”过程。

1）测试过程

➢ 使用 StatefulSet 创建 MySQL 单实例 gxr-oracle-statefulset。

- Scheduler 将 MySQL 单实例调度到集群中的节点“k8s-node3”。
- 通过 sysbench 对该实例制造极高的负载，“k8s-node3”load 飙升，导致“k8s-node3”上的 Kubelet 无法与 API Server 通信，并开始报错。
- Node Controller 启动驱逐。
- StatefulSet 发起重建。
- Scheduler 将 MySQL 实例调度到“k8s-node1”上。
- 新旧 MySQL 实例访问同一个 Volume。
- 数据文件被写坏，旧 MySQL 实例都报错，并无法启动。

2）测试参数

- kube-controller-manager 启动参数。

```
    - --allocate-node-cidrs=true
    - --cluster-cidr=10.244.0.0/16
    - --node-monitor-period=4s
    - --node-monitor-grace-period=32s
    - --pod-eviction-timeout=120s
    image: 10.10.20.3:5000/kube-controller-manager-amd64:v1.7.8
```

- kubelet 启动参数。

```
[root@k8s-master ~]# ps -ef |grep kubelet
root      1768     1  4 07:56 ?        00:17:46 /usr/bin/kubelet --kubeconfig=/etc/kubernetes/kubelet.conf --require-kubeconfig=true --pod-manifest-path=/etc/kubernetes/manifests --allow-privileged=
rue --node-status-update-frequency=8s --network-plugin=cni --cni-conf-dir=/etc/cni/net.d --cni-bin-dir=/opt/cni/bin --cluster-dns=10.96.0.10 --cluster-domain=cluster.local --authorization-mode=Webhoo
k --client-ca-file=/etc/kubernetes/pki/ca.crt --cadvisor-port=0 --cgroup-driver=systemd --pod-infra-container-image=10.10.20.3:5000/gcr.io/google_containers/pause-amd64:3.0
```

- 基于日志，各事件流如下：时间点 December 1st 2017，10:18:05.000（最后一次更新成功应该是 10:17:42.000）。

节点（k8s-node3）启动数据库压力测试，以模拟该节点“假死”，kubelet 与 API Server 出现心跳丢失。

```
▸ December 1st 2017, 10:18:05.000   -   E1201 10:18:05.260359    841 kubelet_node_status.go:357] Error updating node
                                        status, will retry: error getting node "k8s-node3": Get
                                        https://10.10.40.34:6443/api/v1/nodes/k8s-node3: net/http: request canceled
                                        (Client.Timeout exceeded while awaiting headers)

▸ December 1st 2017, 10:18:13.000   -   E1201 10:18:13.260776    841 kubelet_node_status.go:357] Error updating node
                                        status, will retry: error getting node "k8s-node3": Get
                                        https://10.10.40.34:6443/api/v1/nodes/k8s-node3: net/http: request canceled
                                        (Client.Timeout exceeded while awaiting headers)
```

kubelet 日志报错，无法通过 API Server 更新 k8s-node3 状态。通过 API Server 更新集群信息：

```
if kl.kubeClient != nil {
  //Start syncing node status immediately, this may set up things the runtime needsto run.
  gowait.Until(kl.syncNodeStatus, kl.nodeStatusUpdateFrequency, wait.NeverStop)
}
```

定期（node Status Update Frequency）更新对应的节点状态。

nodeStatusUpdateFrequency 默认时间为 10 秒，测试时设置为 8s。

```
obj.NodeStatusUpdateFrequency =metav1.Duration{Duration: 10 * time.Second}
```

更新信息如下。

```
func (kl *Kubelet) defaultNodeStatusFuncs()[]func(*v1.Node) error {
  //initial set of node status update handlers, can be modified by Option's
 withoutError := func(f func(*v1.Node)) func(*v1.Node) error {
    return func(n *v1.Node) error {
      f(n)
```

```
            return nil
        }
    }
    return []func(*v1.Node) error{
        kl.setNodeAddress,
        withoutError(kl.setNodeStatusInfo),
        withoutError(kl.setNodeOODCondition),
        withoutError(kl.setNodeMemoryPressureCondition),
        withoutError(kl.setNodeDiskPressureCondition),
        withoutError(kl.setNodeReadyCondition),
        withoutError(kl.setNodeVolumesInUseStatus),
        withoutError(kl.recordNodeSchedulableEvent),
    }
}
```

通过 kubectl 可以获得节点的信息。

```
Conditions:
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason                       Message
  ----             ------  -----------------                 ------------------                ------                       -------
  OutOfDisk        False   Sat, 02 Dec 2017 13:14:23 +0800   Fri, 01 Dec 2017 10:29:17 +0800   KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure   False   Sat, 02 Dec 2017 13:14:23 +0800   Fri, 01 Dec 2017 10:29:17 +0800   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   Sat, 02 Dec 2017 13:14:23 +0800   Fri, 01 Dec 2017 10:29:17 +0800   KubeletHasNoDiskPressure     kubelet has no disk pressure
  Ready            True    Sat, 02 Dec 2017 13:14:23 +0800   Fri, 01 Dec 2017 10:29:17 +0800   KubeletReady                 kubelet is posting ready status
```

➢ 时间点 December 1st 2017, 10:18:14.000：

Node Controller 发现 k8s-node3 的状态有 32s 没有发生更新。

ready/ outofdisk / diskpressure / memorypressue condition

```
▸ December 1st 2017, 10:18:14.000  k8s_kube-controller-manager_kube-controller-manager-k8s-master_kube-system_d2448e76e66  I1201 02:18:14.134870  1 nodecontroller.go:937] node k8s-node3 hasn't been updated for 32.005088685. Last ready condition is: {Type:Ready Status:True LastHeartbeatTime:2017-12-01 02:17:41 +0000 UTC LastTransitionTime:2017-12-01 00:38:19 +0000 UTC Reason:KubeletReady Message:kubelet is posting ready status}
```

```
▸ December 1st 2017, 10:18:14.000  k8s_kube-controller-manager_kube-controller-manager-k8s-master_kube-system_d2448e76e66  I1201 02:18:14.134910  1 nodecontroller.go:965] node k8s-node3 hasn't been updated for 32.005133245. Last OutOfDisk is: &NodeCondition{Type:OutOfDisk,Status:False,LastHeartbeatTime:2017-12-01 02:17:41 +0000 UTC,LastTransitionTime:2017-12-01 00:38:19 +0000 UTC,Reason:KubeletHasSufficientDisk,Message:kubelet has sufficient disk space available,}
```

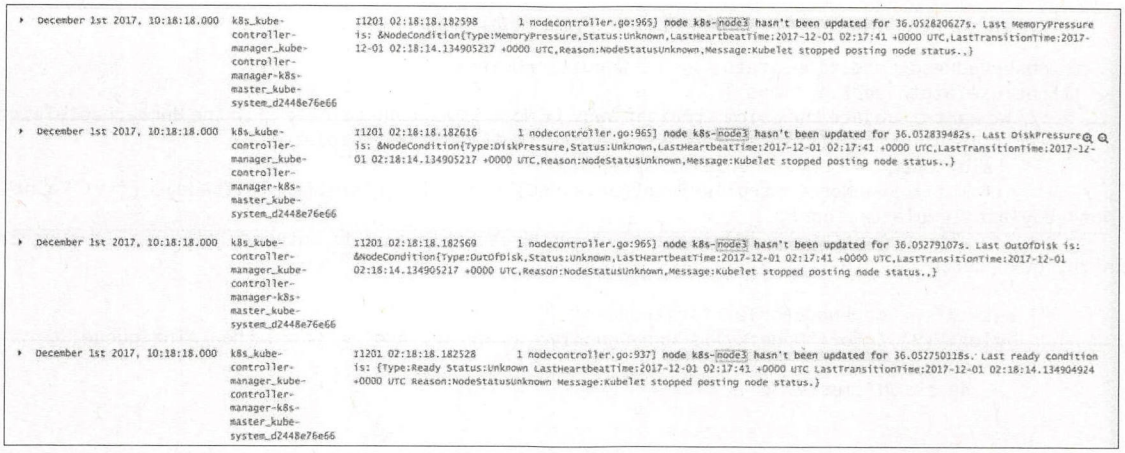
```
▸ December 1st 2017, 10:18:14.000  k8s_kube-controller-manager_kube-controller-manager-k8s-master_kube-system_d2448e76e66  I1201 02:18:14.135077  1 nodecontroller.go:965] node k8s-node3 hasn't been updated for 32.005300371s. Last DiskPressure is: &NodeCondition{Type:DiskPressure,Status:False,LastHeartbeatTime:2017-12-01 02:17:41 +0000 UTC,LastTransitionTime:2017-12-01 00:38:19 +0000 UTC,Reason:KubeletHasNoDiskPressure,Message:kubelet has no disk pressure,}
```

```
▸ December 1st 2017, 10:18:14.000  k8s_kube-controller-manager_kube-controller-manager-k8s-master_kube-system_d2448e76e66  I1201 02:18:14.135049  1 nodecontroller.go:965] node k8s-node3 hasn't been updated for 32.005269137s. Last MemoryPressure is: &NodeCondition{Type:MemoryPressure,Status:False,LastHeartbeatTime:2017-12-01 02:17:41 +0000 UTC,LastTransitionTime:2017-12-01 00:38:19 +0000 UTC,Reason:KubeletHasSufficientMemory,Message:kubelet has sufficient memory available,}
```

将该节点状态更新为 UNKNOWN。

```
Conditions:
  Type             Status    LastHeartbeatTime                 LastTransitionTime                Reason              Message
  ----             ------    -----------------                 ------------------                ------              -------
  OutOfDisk        Unknown   Fri, 01 Dec 2017 10:17:41 +0800   Fri, 01 Dec 2017 10:18:14 +0800   NodeStatusUnknown   Kubelet stopped posting node status.
  MemoryPressure   Unknown   Fri, 01 Dec 2017 10:17:41 +0800   Fri, 01 Dec 2017 10:18:14 +0800   NodeStatusUnknown   Kubelet stopped posting node status.
  DiskPressure     Unknown   Fri, 01 Dec 2017 10:17:41 +0800   Fri, 01 Dec 2017 10:18:14 +0800   NodeStatusUnknown   Kubelet stopped posting node status.
  Ready            Unknown   Fri, 01 Dec 2017 10:17:41 +0800   Fri, 01 Dec 2017 10:18:14 +0800   NodeStatusUnknown   Kubelet stopped posting node status.
```

每隔 NodeMonitorPeriod 继续节点状态是否有更新。



定期（NodeMonitorPeriod）查看一次节点状态。

```
// Incorporate the results of node statuspushed from kubelet to master.
go wait.Until(func() {
  iferr := nc.monitorNodeStatus();  err != nil {
    glog.Errorf("Error monitoring node status: %v", err)
  }
}, nc.nodeMonitorPeriod, wait.NeverStop)
```

NodeMonitorPeriod 默认 5 秒，测试时为 4 秒。

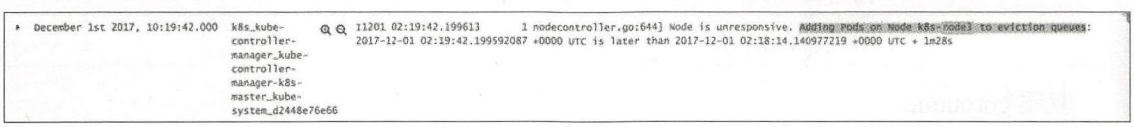
```
NodeMonitorPeriod:                                  metav1.Duration{Duration: 5 * time.Second},
```

当超过 NodeMonitorGracePeriod 时间后，节点状态没有更新，将节点状态设置成 Unknown。

```
if nc.now().After(savedNodeStatus.probeTimestamp.Add(gracePeriod)) {
 // NodeReady condition was last set longer ago than gracePeriod, so update it to Unknown
 // (regardless of its current value) in the master.
 if currentReadyCondition == nil {
    glog.V(2).Infof("node %v is never updated by kubelet", node.Name)
    node.Status.Conditions = append(node.Status.Conditions, v1.NodeCondition{
       Type:               v1.NodeReady,
       Status:             v1.ConditionUnknown,
       Reason:             "NodeStatusNeverUpdated",
       Message:            fmt.Sprintf("Kubelet never posted node status."),
       LastHeartbeatTime:  node.CreationTimestamp,
       LastTransitionTime: nc.now(),
    })
 } else {
    glog.V(4).Infof("node %v hasn't been updated for %+v. Last ready condition is: %+v",
       node.Name, nc.now().Time.Sub(savedNodeStatus.probeTimestamp.Time), observedReadyCondition)
    if observedReadyCondition.Status != v1.ConditionUnknown {
       currentReadyCondition.Status = v1.ConditionUnknown
       currentReadyCondition.Reason = "NodeStatusUnknown"
       currentReadyCondition.Message = "Kubelet stopped posting node status."
       // LastProbeTime is the last time we heard from kubelet.
       currentReadyCondition.LastHeartbeatTime = observedReadyCondition.LastHeartbeatTime
       currentReadyCondition.LastTransitionTime = nc.now()
    }
 }
```

➢ 时间点为 December 1st 2017，10:19:42.000。

刚好过去 podEvictionTimeout，将该节点添加到驱逐队列中。



在 podEvictionTimeout 后，为该节点上 Pods 开始驱逐。

```
    if observedReadyCondition.Status == v1.ConditionUnknown {
     if nc.useTaintBasedEvictions {
        // We want to update the taint straight away if Node is already tainted with the UnreachableTaint
        if taintutils.TaintExists(node.Spec.Taints, NotReadyTaintTemplate) {
           taintToAdd := *UnreachableTaintTemplate
           if !util.SwapNodeControllerTaint(nc.kubeClient, []*v1.Taint{&taintToAdd}, []*v1.Taint
{NotReadyTaintTemplate}, node) {
              glog.Errorf("Failed to instantly swap UnreachableTaint to NotReadyTaint. Will try again
in the next cycle.")
           }
        } else if nc.markNodeForTainting(node) {
           glog.V(2).Infof("Node %v is unresponsive as of %v. Adding it to the Taint queue.",
              node.Name,
              decisionTimestamp,
           )
        }
     } else {
        if decisionTimestamp.After(nc.nodeStatusMap[node.Name].probeTimestamp.Add(nc.podEviction-
Timeout)) {
           if nc.evictPods(node) {
              glog.V(2).Infof("Node is unresponsive. Adding Pods on Node %s to eviction queues: %v
is later than %v + %v",
                 node.Name,
                 decisionTimestamp,
                 nc.nodeStatusMap[node.Name].readyTransitionTimestamp,
                 nc.podEvictionTimeout-gracePeriod,
              )
           }
        }
     }
    }
```

放到驱逐数组中。

```
    // evictPods queues an eviction for the provided node name, and returns false if the node is already
    // queued for eviction.
    func (nc *Controller) evictPods(node *v1.Node) bool {
     nc.evictorLock.Lock()
     defer nc.evictorLock.Unlock()
     return nc.zonePodEvictor[utilnode.GetZoneKey(node)].Add(node.Name, string(node.UID))
    }
```

时间点为 December 1st 2017, 10:19:42.000。

开始驱逐。

December 1st 2017, 10:19:42	k8s_kube-controller-manager_kube-controller-manager-k8s-master_kube-system_d2448e76e66	I1201 02:19:42.284926	1 controller_utils.go:89] Starting deletion of pod default/gxr1-oracle-statefulset-0

December 1st 2017, 10:19:42.000	k8s_kube-controller-manager_kube-controller-manager-k8s-master_kube-system_d2448e76e66	I1201 02:19:42.276978	1 event.go:218] Event(v1.ObjectReference{Kind:"Node", Namespace:"", Name:"k8s-node3", UID:"cd1eb2d6-bc4d-11e7-b94c-525400e717a6", APIVersion:"", ResourceVersion:"", FieldPath:""}): type: 'Normal' reason: 'DeletingAllPods' Node k8s-node3 event: Deleting all Pods from Node k8s-node3.

December 1st 2017, 10:19:42.000	k8s_kube-controller-manager_kube-controller-manager-k8s-master_kube-system_d2448e76e66	I1201 02:19:42.293174	1 disruption.go:378] deletePod called on pod "gxr1-oracle-statefulset-0"

驱逐 goroutine。

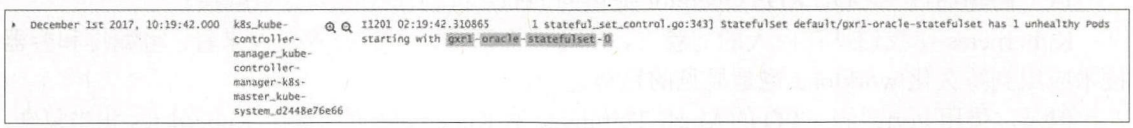
```
    if nc.useTaintBasedEvictions {
     // Handling taint based evictions. Because we don't want a dedicated logic in TaintManager for
NC-originated
     // taints and we normally don't rate limit evictions caused by taints, we need to rate limit adding
taints.
     go wait.Until(nc.doNoExecuteTaintingPass, scheduler.NodeEvictionPeriod, wait.NeverStop)
    } else {
     // Managing eviction of nodes:
     // When we delete pods off a node, if the node was not empty at the time we then
     // queue an eviction watcher. If we hit an error, retry deletion.
     go wait.Until(nc.doEvictionPass, scheduler.NodeEvictionPeriod, wait.NeverStop)
    }
```

通过删除 Pods 的方式驱逐。

```
    func (nc *Controller) doEvictionPass() {
     nc.evictorLock.Lock()
     defer nc.evictorLock.Unlock()
     for k := range nc.zonePodEvictor {
        // Function should return 'false' and a time after which it should be retried, or 'true' if
it shouldn't (it succeeded).
        nc.zonePodEvictor[k].Try(func(value scheduler.TimedValue) (bool, time.Duration) {
           node, err := nc.nodeLister.Get(value.Value)
           if apierrors.IsNotFound(err) {
              glog.Warningf("Node %v no longer present in nodeLister!", value.Value)
           } else if err != nil {
              glog.Warningf("Failed to get Node %v from the nodeLister: %v", value.Value, err)
           } else {
              zone := utilnode.GetZoneKey(node)
              evictionsNumber.WithLabelValues(zone).Inc()
           }
           nodeUID, _ := value.UID.(string)
            remaining, err := util.DeletePods(nc.kubeClient, nc.recorder, value.Value, nodeUID,
nc.daemonSetStore)
           if err != nil {
              utilruntime.HandleError(fmt.Errorf("unable to evict node %q: %v", value.Value, err))
              return false, 0
           }
           if remaining {
              glog.Infof("Pods awaiting deletion due to Controller eviction")
           }
           return true, 0
        })
     }
    }
```

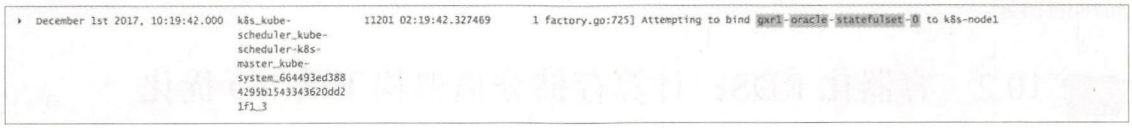
➢ 时间点 December 1st 2017，10:19:42.000。

StatefulSet controller 发现 default/gxr1-oracle-statefulset 状态异常。



时间点 December 1st 2017, 10:19:42.000。

Scheduler 将 Pod 调度到 k8s-node1。



这样旧的 MySQL 实例在 k8s-node3 上，Kubernetes 又将新的实例调度到 k8s-node1。

两个数据库实例写同一份数据文件，data corruption，两个节点都无法启动。

实例启动报错，日志为：

```
    2017-12-01 10:19:47 5628 [Note] mysqld (mysqld 5.7.19-log) starting as process 963 ...
    2017-12-01 10:19:47 5628 [Note] InnoDB: PUNCH HOLE support available
    2017-12-01 10:19:47 5628 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
    2017-12-01 10:19:47 5628 [Note] InnoDB: Uses event mutexes
    2017-12-01 10:19:47 5628 [Note] InnoDB: GCC builtin __atomic_thread_fence() is used for memory
barrier
    2017-12-01 10:19:47 5628 [Note] InnoDB: Compressed tables use zlib 1.2.3
    2017-12-01 10:19:47 5628 [Note] InnoDB: Using Linux native AIO
    2017-12-01 10:19:47 5628 [Note] InnoDB: Number of pools: 1
    2017-12-01 10:19:47 5628 [Note] InnoDB: Using CPU crc32 instructions
    2017-12-01 10:19:47 5628 [Note] InnoDB: Initializing buffer pool, total size = 3.25G,
instances = 2, chunk size = 128M
    2017-12-01 10:19:47 5628 [Note] InnoDB: Completed initialization of buffer pool
    2017-12-01 10:19:47 5628 [Note] InnoDB: If the mysqld execution user is authorized, page cleaner
thread priority can be changed. See the man page of setpriority().
    2017-12-01 10:19:47 5628 [Note] InnoDB: Highest supported file format is Barracuda.
    2017-12-01 10:19:47 5628 [Note] InnoDB: Log scan progressed past the checkpoint lsn 406822323
    2017-12-01 10:19:47 5628 [Note] InnoDB: Doing recovery: scanned up to log sequence number 406823190
    2017-12-01 10:19:47 5628 [Note] InnoDB: Database was not shutdown normally!
    2017-12-01 10:19:47 5628 [Note] InnoDB: Starting crash recovery.
    2017-12-01 10:19:47 5669 [Note] InnoDB: Starting an apply batch of log records to the database...
    InnoDB: Progress in  percent: 89 90 91 92 93 94 95 96 97 98 99
    2017-12-01 10:19:47 5669 [Note] InnoDB: Apply batch completed
    2017-12-01 10:19:47 5669 [Note] InnoDB: Last MySQL binlog file position 0 428730, file name
mysql-bin.000004
    2017-12-01 10:19:47 5669 [Note] InnoDB: Removed temporary tablespace data file: "ibtmp1"
    2017-12-01 10:19:47 5669 [Note] InnoDB: Creating shared tablespace for temporary tables
    2017-12-01 10:19:47 5669 [Note] InnoDB: Setting file './ibtmp1' size to 12 MB. Physically writing
the file full;  Please wait ...
    2017-12-01 10:19:47 5669 [Note] InnoDB: File './ibtmp1' size is now 12 MB.
    2017-12-01 10:19:47 5669 [Note] InnoDB: 96 redo rollback segment(s) found. 96 redo rollback
segment(s) are active.
    2017-12-01 10:19:47 5669 [Note] InnoDB: 32 non-redo rollback segment(s) are active.
    2017-12-01 10:19:47 5669 [Note] InnoDB: Waiting for purge to start
    2017-12-01 10:19:47 0x7fcb08928700  InnoDB: Assertion failure in thread 140509998909184 in file
trx0purge.cc line 168
    InnoDB: Failing assertion: purge_sys->iter.trx_no <= purge_sys->rseg->last_trx_no
    InnoDB: We intentionally generate a memory trap.
    InnoDB: Submit a detailed bug report to http://bugs.mysql.com.
    InnoDB: If you get repeated assertion failures or crashes, even
    InnoDB: immediately after the mysqld startup, there may be
    InnoDB: corruption in the InnoDB tablespace. Please refer to
    InnoDB: http://dev.mysql.com/doc/refman/5.7/en/forcing-innodb-recovery.html
    InnoDB: about forcing recovery.
    10:19:47 5669 - mysqld got signal 6 ;
```

以上问题通过 WOQU RDS Operator 提供的 Fence 机制已经得到有效解决。

Kubernetes 使我们站在巨人的肩膀上，从各大互联网公司的技术发展来看，将编排和容器技术应用到持久化 workload 也是显见的趋势之一。

但是，借用 Portworx CEO 的 Murli Thirumale 对 Kubernetes 的预测：Kubernetes 相当复杂。Kubernetes 被拥趸们冠以“优雅”的头衔，但优雅并不意味着简单。弦论是优雅的，但是理解它需要付出极大的努力。Kubernetes 也一样，使用 Kubernetes 构建和运行应用程序并不是一个简单的命题。

10.2 容器化 RDS：计算存储分离架构下的 I/O 优化

在基于 Kubernetes 和 Docker 构建的私有 RDS 中，普遍采用了计算存储分离架构。该架构的优势明显，但对于数据库类 Latency Sensitive 应用而言，I/O 性能问题无法回避，下面分享

针对 MySQL 做的优化及优化后的收益。

10.2.1　计算存储分离架构

存储层由分布式文件系统组成，以 Provisoner 的方式集成到 Kubernetes。在我们看来，计算存储分离的最大优势在于：

将有状态的数据下沉到存储层，这使得 RDS 在调度时无须感知计算节点的存储介质，只需调度到满足计算资源要求的 Node。数据库实例启动时，只需在分布式文件系统挂载 mapping 的 volume 即可，可以显著地提高数据库实例的部署密度和计算资源利用率。

其他的好处还有很多，譬如架构更清晰、扩展更方便、问题定位更简单等，这里不再赘述。

10.2.2　计算存储分离架构的缺点

如图 10-6 所示，相比本地存储，网络开销会成为 I/O 开销的一部分，会带来两个很明显的问题：

- 数据库是 Latency Sensitive 型应用，网络延时会极大地影响数据库能力（QPS，TPS）。
- 在高密度部署的场景，网络带宽会成为瓶颈，可能导致计算资源和存储资源利用不充分。

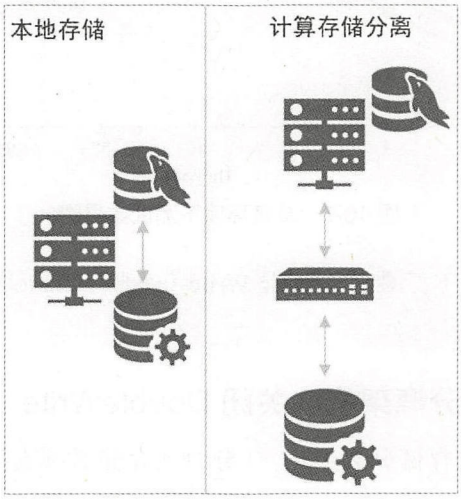


图 10-6　本地存储与计算存储分离示意图

其实还有一个极其重要的问题，由于 Kubernetes 本身没有提供 Voting 服务和类似 Oracle Rac 的 Fence 机制。在计算存储分离架构下，当集群发生脑裂，并触发 Node Controller 和 Kubelet 的驱逐机制时，可能会出现多个数据库实例同时访问一份数据文件，导致数据坏块的情况，数据的损失对用户而言是不可估量也是不可忍受的。

沃趣在 Kubernetes 1.7.8 版本下使用 Oracle、MySQL 都可以百分之百地复现这个场景，通过在 Kubernetes 上添加 Fence 机制，已解决该问题。下面结合 MySQL 的特性进行有针对性的优化。

10.2.3　DoubleWrite

在 MySQL 中，我们首先想到了 DoubleWrite。从官方解释来看，简单地说，DoubleWrite 的实现是防止数据页写入时发生故障导致页损坏（Partial Write），所以每次写数据文件时都要将一份数据写到共享表空间中，当启动时发现数据页 Checkum 校验不正确时会使用共享表空

间中的副本进行恢复，从 DoubleWrite 实现来看，这部分会产生一定量的 I/O。所以，最好的优化是减少 I/O，在底层存储介质或文件系统支持 Atomic Write 的前提下，可以关闭 MySQL 的 DoubleWrite，以减少 I/O。

10.2.4 单机架构：关闭 DoubleWrite

MariaDB 已支持该功能（底层存储介质需支持 Atomic Write），并在单机环境做了相关测试，数据如图 10-7 所示。

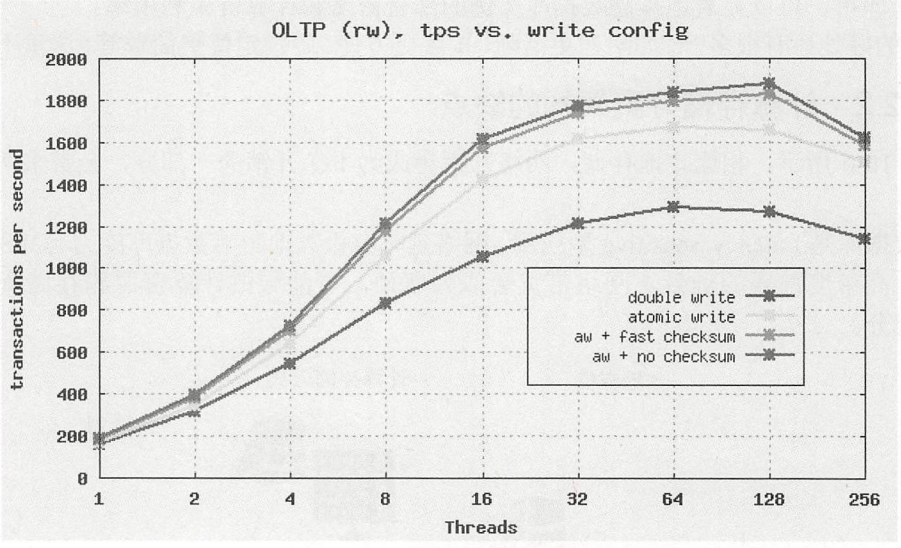


图 10-7 单机环境下测试效果图

结论：在单机环境下，启用 Atomic Write（关闭 DoubleWrite）能立即带来 30% 左右的写性能改善。

10.2.5 计算存储分离架构：关闭 DoubleWrite

重点是需要测试在计算存储分离架构（分布式存储必须支持 Atomic Write）下，关闭 DoubleWrite Buffer 的收益。

1. 测试场景

- 采用 Sysbench 模拟 OLTP 模型（跟 MariaDB 相同）。
- 数据库版本选择 MySQL 5.7.19（测试时的最新版本）。
- 由本地存储改为分布式文件系统。
- 测试数据量，数据文件负载模型分为 10GB 和 100GB。

2. 测试结果

1）10GB 数据量 Sysbench 指标如表 10-1 所示。

表 10-1 10GB 数据量 Sysbench 指标

指标类型	线程个数	表数量	数据量	测试时长/min	平均 TPS	平均 QPS	响应时间（95%）/ms
oltp 开双写	256	8	500W	10	5632	112643	73.13
oltp 关双写	256	8	500W	10	5647	112959	86.00

10GB 数据量分布式文件系统指标如图 10-8 所示。

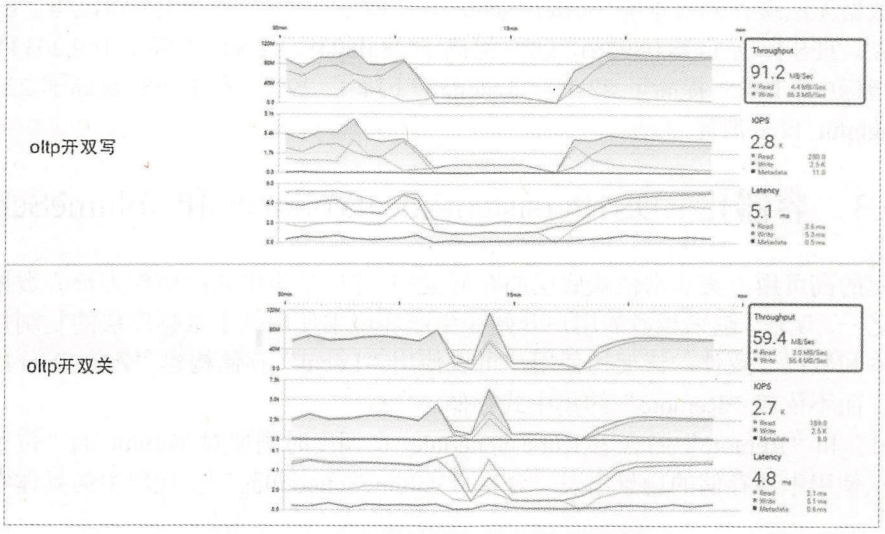


图 10-8　10GB 数据量分布式文件系统指标

在计算存储分离架构下，启用 Atomic Write（关闭 DoubleWrite）。对于 10GB 的数据量，因为大部分数据已经缓存到数据库 buffer cache 中，所以在 I/O 不是瓶颈的情况下，Sysbench 指标的提升不明显。TPS 提高了 0.2656%，QPS 提高了 0.2797%，RST 提高了 14.9651%；分布式文件系统指标中 Throughput 下降了 53%，显著优化了网络带宽。

2）100GB 数据量 Sysbench 指标如表 10-2 所示。

表 10-2　100GB 数据量 Sysbench 指标

指标类型	线程个数	表数量	数据量	测试时长/min	平均 TPS	平均 QPS	响应时间(95%)/ms
oltp 开双写	256	8	500W	10	2260	45202	227.40
oltp 关双写	256	8	500W	10	2519	50394	277.21

10GB 数据量分布式文件系统指标如图 10-9 所示。

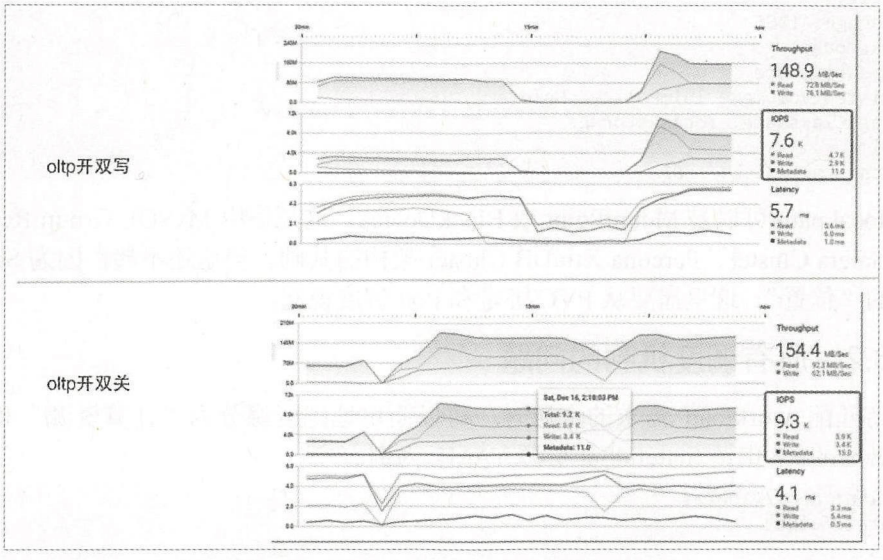


图 10-9　100GB 数据量分布式文件系统指标

在计算存储分离架构下，启用 Atomic Write（关闭 DoubleWrite），对于 100GB 数据量，因为大部分数据无法缓存到数据库 buffer cache 中，所以在 I/O 是瓶颈的情况下，Sysbench 指标提升明显。TPS 提高了 28.0892%，QPS 提高了 28.0893%，RST 下降了 169.2033%；分布式文件系统指标中，IOPS 提高了 22.3%，Latency 下降了 39%。在 IOPS 提高了 22.3%的情况下，Throughput 仅多消耗 3.6%。

10.3 容器化 RDS：PersistentLocalVolumes 和 VolumeScheduling

数据库的高可用方案非常依赖底层的存储架构，这也是集中式存储作为核心数据库业务标配的原因之一。现在，越来越多的用户开始在生产环境中使用基于数据库层的复制技术来保障数据多副本和数据一致性，该架构使用户可以使用“Local”存储构建“Zero Data Lost”的高可用集群，而不依赖“Remote”的集中式存储。

“Local”和“Remote”隐含着 Kube-Scheduler 在调度时需要对 Volume 的“位置”可见。本文尝试从使用本地存储的场景出发，分享“VolumeScheduling”在代码中的具体实现和场景局限。

10.3.1 本地卷

相比“Remote”的卷，本地卷具有如下特点：

- 更好地利用本地高性能介质（SSD，Flash）提升数据库服务能力 QPS/TPS。
- 更闭环的运维成本，现在越来越多的数据库支持基于 Replicated 的技术实现数据多副本和数据一致性（比如 MySQL Group Replication、MariaDB Galera Cluster、Percona XtraDB Cluster），DBA 可以处理所有问题，而不再依赖存储工程师或 SA 的支持。

在 Kubernetes 1.9 版本之后，可以通过 Feature Gate“PersistentLocalVolumes”使用本地卷。

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: local-pv
spec:
   capacity:
     storage: 10Gi
   accessModes:
   - ReadWriteOnce
   persistentVolumeReclaimPolicy: Delete
   storageClassName: local-storage
   local:
     path: /mnt/disks/ssd1
```

目前 local.path 可以是 MountPoint 或 BlockDevice。这是使用 MySQL Group Replication 、MariaDB Galera Cluster、Percona XtraDB Cluster 架构的基础。但这还不够，因为 Scheduler 并不感知卷的“位置”。这里需要从 PVC 绑定和 Pod 调度说起。

10.3.2 原有调度机制的问题

当申请匹配 workload 需求的资源时，可以简单地把资源分为“计算资源”和“存储资源”。以 Kubernetes 申请 StatefulSet 资源 YAML 为例。

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
 name: mysql-5.7
spec:
```

```
  replicas: 1
  template:
    metadata:
      name: mysql-5.7
    spec:
      containers:
        name: mysql
        resources:
          limits:
            cpu: 5300m
            memory: 5Gi
        volumeMounts:
        - mountPath: /var/lib/mysql
          name: data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes:
      - ReadWriteOnce
      resources:
        requests:
          storage: 10Gi
```

YAML 中定义了 Pod 对“计算”和“存储”资源的要求，随后 StatefulSet Controller 创建需要的 PVC 和 Pod。

```
func (spc *realStatefulPodControl) CreateStatefulPod(set *apps.StatefulSet, pod *v1.Pod) error {
    // Create the Pod's PVCs prior to creating the Pod
    if err := spc.createPersistentVolumeClaims(set, pod); err != nil {
        spc.recordPodEvent("create", set, pod, err)
        return err
    }
    // If we created the PVCs attempt to create the Pod
    _, err := spc.client.CoreV1().Pods(set.Namespace).Create(pod)
    // sink already exists errors
    if apierrors.IsAlreadyExists(err) {
        return err
    }
    spc.recordPodEvent("create", set, pod, err)
    return err
}
```

10.3.3　PVC 绑定

Pod 借用 PVC 描述需要的存储资源，PVC 是 PV 的抽象，就像 VFS 是 Linux 对具体文件系统的抽象一样。所以，在 PVC 创建之后，还需要将 PVC 与卷绑定，也即是 PV。PersistentVolume Controller 会遍历现有 PV 和可以动态创建的 StorageClass（比如 NFS、Ceph、EBS），找到满足条件（访问权限、容量等）进行绑定。

10.3.4　Pod 调度

Scheduler 基于资源要求找到匹配的节点，以过滤和打分的方式选出“匹配度”最高的 Node，流程大致如图 10-10 所示。

- PVC 绑定在 Pod 调度之前，PersistentVolume Controller 不会等待 Scheduler 调度结果。在 StatefulSet 中，PVC 先于 Pod 创建，所以 PVC/PV 绑定可能在 Pod 调度之前完成。
- Scheduler 不感知卷的“位置”，仅考虑存储容量、访问权限、存储类型，还有第三方 CloudProvider 上的限制，比如在 AWS、GCE、Azure 上使用硬盘数量的限制。

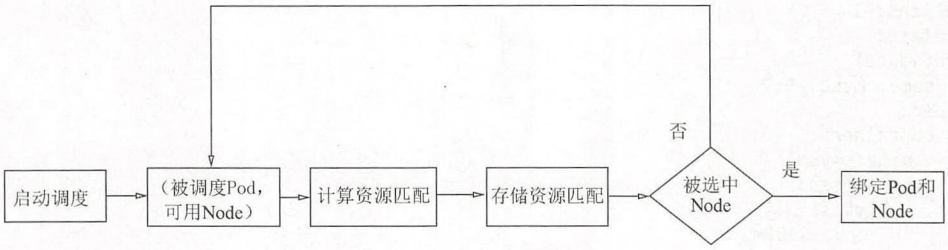


图 10-10　Pod 调度流程

当应用对卷的“位置”有要求时，比如使用本地卷，可能出现 Pod 被 Scheduler 调度到 NodeB，但 PersistentVolume Controller 绑定了在 NodeD 上的本地卷，以致 PV 和 Pod 不在一个节点，如图 10-111 所示。

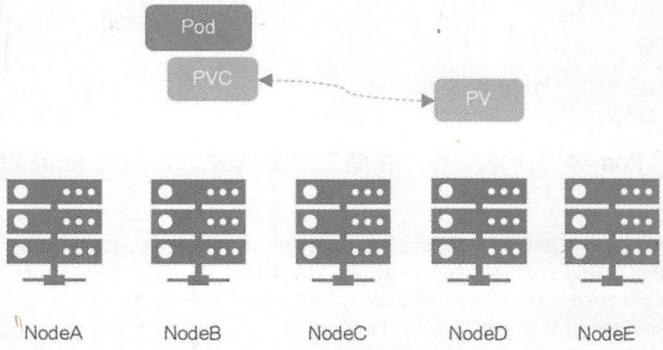


图 10-11　PV 和 Pod 节点不一致

不仅仅是本地卷，如果对存储“位置”（如 Rack、Zone）有要求，都会有类似问题。

好比 Pod 作为下属，它实现自身价值的核心资源来自两个上级 Scheduler 和 PersistentVolume Controller，但是这两个上级从来不沟通，甚至出现矛盾。作为下属，要解决这个问题，无非如下几种选择：

1）尝试让两个老板沟通。

2）站队，挑一个老板，只听其中一个的指挥。

3）辞职。

Kubernetes 做出了“正常人”的选择：站队。如果 Pod 使用的 Volume 对“位置”有要求（又叫 Topology-Aware Volume），通过延时绑定（DelayBinding），使 PersistentVolume Controller 不再参与，PVC 绑定的工作全部由 Scheduler 完成。

在通过代码了解特性“VolumeScheduling”的具体实现时，还可以先思考如下几个问题：

- 如何标记 Topology-Aware Volume。
- 如何让 PersistentVolume Controller 不再参与，同时不影响原有流程。

1. Feature：VolumeScheduling

Kubernetes 在卷管理中通过策略 VolumeScheduling 重构 Scheduler，以支持 Topology-Aware Volume，步骤大致如下：

- 预分配使用本地卷的 PV。
- 通过 NodeAffinity 方式标记 Topology-Aware Volume 和“位置”信息。

```
"volume.alpha.kubernetes.io/node-affinity": '{
          "requiredDuringSchedulingIgnoredDuringExecution": {
             "nodeSelectorTerms": [
                { "matchExpressions": [
                   { "key": "kubernetes.io/hostname",
                     "operator": "In",
                     "values": ["Node1"]
                   }
                ]}
             ]}
          }'
```

- 创建 StorageClass，通过 StorageClass 间接标记 PVC 的绑定需要延后（绑定延时）。
- 标记该 PVC 需要延后到 Node 选择出来之后再绑定：创建 StorageClass“X”（无需 Provisioner），并设置 StorageClass.VolumeBindingMode = VolumeBindingWaitForFirst-Consumer；PVC.StorageClass 设置为 X。

依照原有流程创建 PVC 和 Pod，但对于需要延时绑定的 PVC，PersistentVolume Controller 不再参与。

通过 PVC.StorageClass，PersistentVolume Controller 得知 PVC 是否需要延时绑定。

```
return *class.VolumeBindingMode == storage.VolumeBindingWaitForFirstConsumer
```

如需延时绑定，则说明什么也没有做。

```
if claim.Spec.VolumeName == "" {
      // User did not care which PV they get.
      delayBinding, err := ctrl.shouldDelayBinding(claim)
                    ….
                    switch {
          case delayBinding:
                                do nothing
```

执行原有 Predicates 函数；执行添加 Predicate 函数 CheckVolumeBinding 校验候选 Node 是否满足 PV 物理拓扑（主要逻辑由 FindPodVolumes 提供）。

已绑定 PVC：对应 PV.NodeAffinity 需匹配候选 Node，否则该节点需要 pass。

未绑定 PVC：该 PVC 是否需要延时绑定，如需要，遍历未绑定 PV，其 NodeAffinity 是否匹配候选 Node。如满足，记录 PVC 和 PV 的映射关系到缓存 bindingInfo 中，留待节点最终选出来之后进行最终的绑定。

当以上都不满足时：PVC.StorageClass 是否可以动态地创建 Topology-Aware Volume（又叫 Topology-aware dynamic provisioning）。

执行原有 Priorities 函数。执行添加 Priority 函数 PrioritizeVolumes。Volume 容量匹配越高越好，避免本地存储资源浪费。Scheduler 选出 Node。由 Scheduler 进行 API update，完成最终的 PVC/PV 绑定（异步操作，时间具有不确定性，可能失败）。从缓存 bindingInfo 中获取候选 Node 上 PVC 和 PV 的绑定关系，并通过 API 完成实际的绑定。如果需要 StorageClass 动态创建，被选出 Node 将被赋值给 StorageClass.topologyKey，作为 StorageClass 创建 Volume 的拓扑约束，该功能的实现还在讨论中。绑定被调度 Pod 和 Node。

Scheduler 的流程调整如图 10-12 所示（依据 Kubernetes 1.9 代码）。

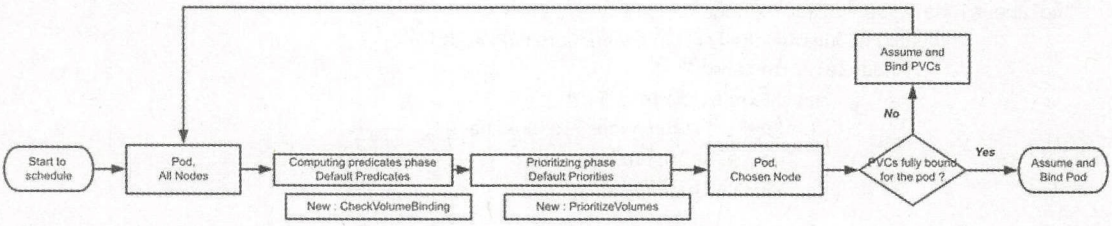


图 10-12　Scheduler 的流程调整

从代码层面，对 Controller Manager 和 Scheduler 都有改造，如图 10-13 所示（依据 Kubernetes 1.9 代码）。

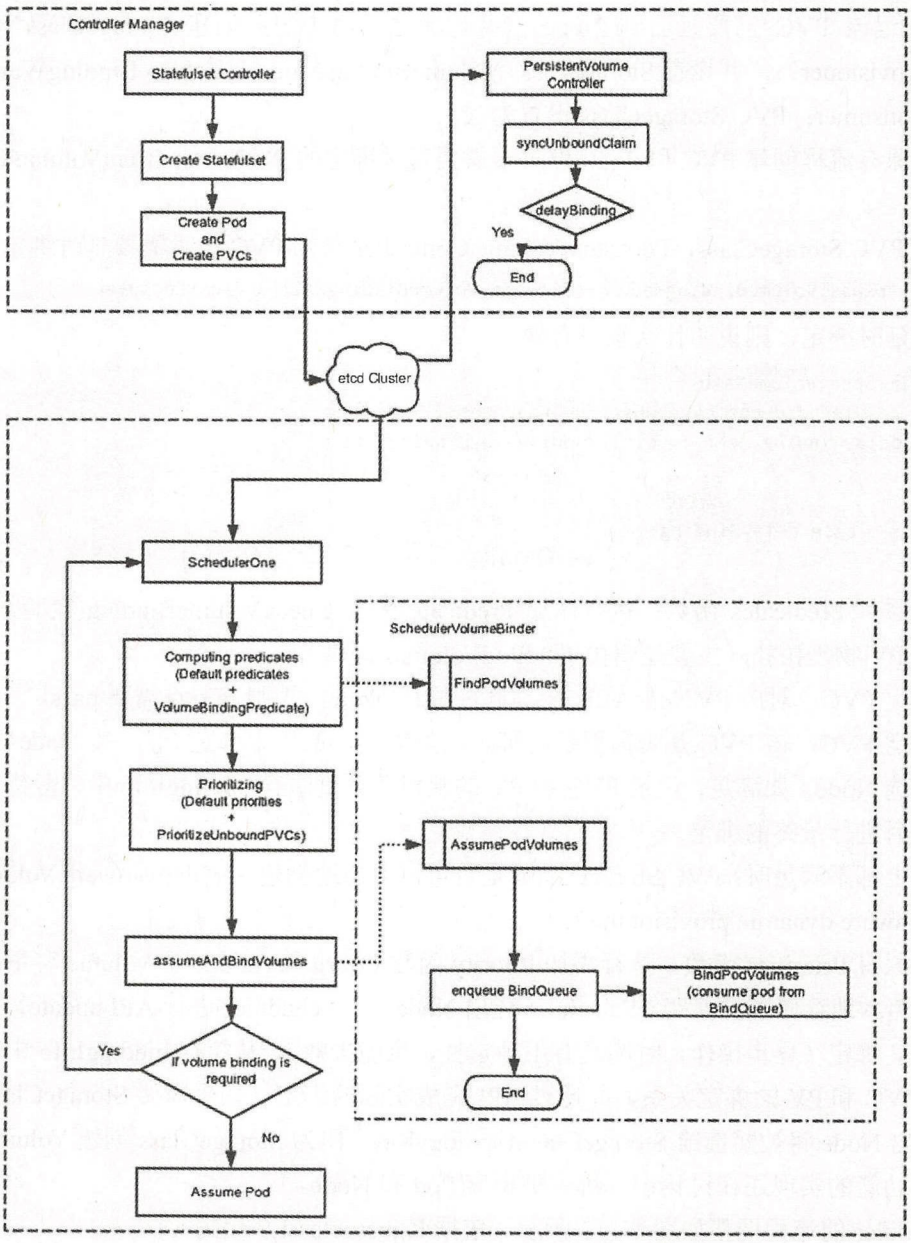


图 10-13　对 Controller Manager 和 Scheduler 的改造

2. 例子

先运行一个例子，使用预分配的方式创建使用本地存储的 PV。为了使用本地存储，需要启动 FeatureGate:PersistentLocalVolumes 支持本地存储，1.9 是 alpha 版本，1.10 是 beta 版，默认开启。

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: local-pv
 annotations:
       "volume.alpha.kubernetes.io/node-affinity": '{
           "requiredDuringSchedulingIgnoredDuringExecution": {
               "nodeSelectorTerms": [
                   { "matchExpressions": [
                       { "key": "kubernetes.io/hostname",
                         "operator": "In",
                         "values": ["k8s-node1-product"]
                       }
                   ]}
                ]}
            }'
spec:
   capacity:
     storage: 10Gi
   accessModes:
   - ReadWriteOnce
   persistentVolumeReclaimPolicy: Delete
   storageClassName: local-storage
   local:
     path: /mnt/disks/ssd1
```

创建 Storage Class。

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
 name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

创建使用本地存储的 Statefulset（仅列出关键信息）。

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
 name: mysql-5.7
spec:
 replicas: 1
 template:
   metadata:
     name: mysql-5.7
   spec:
     containers:
       name: mysql
       resources:
         limits:
           cpu: 5300m
           memory: 5Gi
       volumeMounts:
       - mountPath: /var/lib/mysql
         name: data
 volumeClaimTemplates:
 - metadata:
     annotations:
       volume.beta.kubernetes.io/storage-class: local-storage
```

```
    name: data
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 10Gi
```

该 Statefulset 的 Pod 将会调度到 k8s-node1-product，并使用本地存储“local-pv”。

3. “PersistentLocalVolumes”和“VolumeScheduling”的局限

在具体部署时，使用局限需要考虑：

- 资源利用率降低。一旦本地存储使用完，即使 CPU、Memory 剩余再多，该节点也无法提供服务。
- 需要做好本地存储规划，譬如每个节点 Volume 的数量、容量等，就像原来使用存储时需要把 LUN 规划好一样，在一个大规模运行的环境，存在落地难度。

高可用风险需要考虑当 Pod 调度到某个节点后，将会跟该节点产生亲和，一旦 Node 发生故障，Pod 不能调度到其他节点，只能等待该节点恢复。能做的就是等待“Node 恢复”，如果部署 3 节点 MySQL 集群，再挂一个 Node，集群将无法提供服务，能做的还是“等待 Node 恢复”。这么设计也是合理的，社区认为该 Node 为 Stateful 节点，Pod 被调度到其他可用 Node 会导致数据丢失。而且还要思考，更好地利用本地高性能介质（SSD、Flash）提升数据库服务能力 QPS/TPS，真的成立吗？数据库是 I/O 延时敏感型应用，同时它也极度依赖系统的平衡性，基于 Replicated 架构的数据库集群对集群网络的要求会很高，一旦网络成为瓶颈影响到数据的 sync replication，都会极大地影响数据库集群服务能力。目前，Kubernetes 的网络解决方案还无法提供高吞吐、低延时的网络能力。

当然，可以解决“等待 Node 恢复”的问题。以 MySQL Group Replication / MariaDB Galera Cluster / Percona XtraDB Cluster 架构为例，完全可以在此基础上结合 Kubernetes 现有的 Control-Plane 做进一步优化。

- Node 不可用后，等待阈值超时，以确定 Node 无法恢复。
- 如确认 Node 不可恢复，删除 PVC，通过解除 PVC 和 PV 绑定的方式，解除 Pod 和 Node 的绑定。
- Scheduler 将 Pod 调度到其他可用 Node，PVC 重新绑定到可用 Node 的 PV。
- Operator 查找 MySQL 最新备份，复制到新的 PV。
- MySQL 集群通过增量同步方式恢复实例数据。
- 增量同步变为实时同步，MySQL 集群恢复。

Kubernetes 并不能包治百病，了解边界是使用的开始，泛泛而谈 Cloud Native 无法获得任何收益，在特定的场景和领域，很多问题还需要自己解决。建议如无特别必要，尽量不选用 Local Volume。

10.4 容器化 RDS：借助 CSI 扩展 Kubernetes 存储能力

RDS 并不是新生事物，新鲜的是通过容器技术和容器编排技术构建 RDS。对金融客户而言，他们有强烈拥抱 Docker 和 Kubernetes 的愿望，但可用性是尝试新技术的前提。存储是持久化应用的关键资源，它是 Monolithic 应用走向 Cloud Native 架构的关键。Kubernetes 存储子系统已经非常强大，但还欠缺一些基础功能，譬如支持 Expand Volume（部分 Storage Vendor 支持）和

SnapShot。本文尝试从沃趣的实现分享如下几方面的内容：

- 现有 Kubernetes 存储插件系统问题。
- Container Storage Interface（CSI）。
- 基于 CSI 和分布式文件系统实现在 MySQL 的 Volume 动态扩展。
- 对 CSI 的展望。

名词说明如表 10-3 所示。

表 10-3　名词说明

原　名	简　称
容器编排系统	CO.
存储提供者	SP.
存储插件接口	Volume Plugin Interface
存储驱动	Volume Driver
容器存储接口 Container Storage Interface	CSI

10.4.1　现有 Kubernetes 存储插件系统问题

可供选的容器编排系统（后面简称 CO.）有很多，除去 Kubernetes 还有 Mesos、Swarm、Cloud Foundry。以 Kubernetes 为例，其通过 PersistentVolume 抽象对以下存储的支持：GCEPersistentDisk、AWSElasticBlockStore、AzureFile、AzureDisk、FC（Fibre Channel）、FlexVolume、Flocker、NFS、iSCSI、RBD（Ceph Block Device）、CephFS、Cinder（OpenStack block storage）、GlusterFS、VsphereVolume、Quobyte Volumes、HostPath 、VMware Photon、Portworx Volumes、ScaleIO Volumes、StorageOS。

不可谓不丰富，Kuberentes 以插件化的方式支持存储厂商（下文简称“SP.”）。SP.通过实现 Kubernetes 存储插件接口（下文简称“Volume Plugin Interface”）的方式提供自己的存储驱动（下文简称“Volume Driver”）：VolumePlugin、PersistentVolumePlugin、DeletableVolumePlugin、ProvisionableVolumePlugin、ExpandableVolumePlugin、Provisioner、Deleter。

以上接口并不需要全部实现，其中 VolumePlugin 是必须实现的接口。

系统架构如图 10-14 所示。

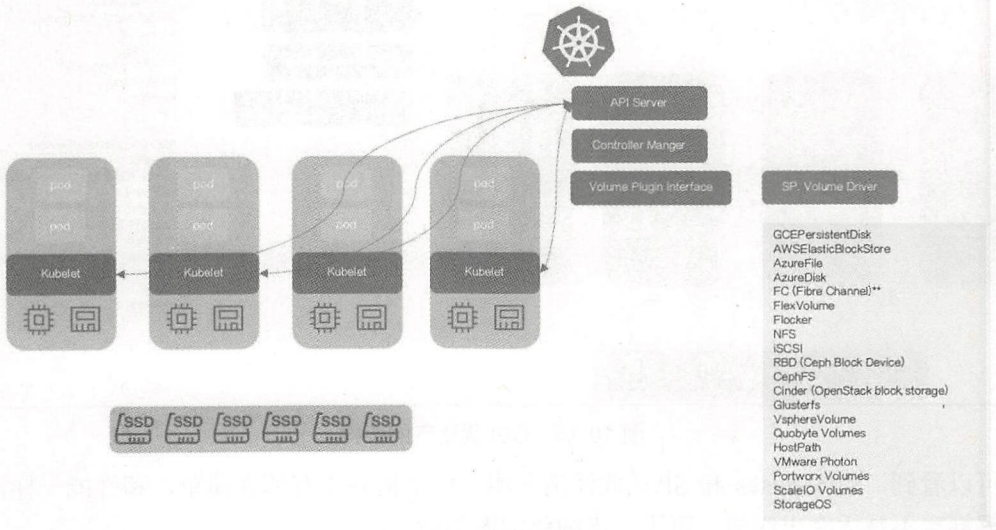


图 10-14　系统架构

这种方式为 Kubernetes 提供了丰富的存储支持列表。但是在具体实现上，SP. Volume Driver 代码也在 Kubernetes 代码仓库中（又叫 in-tree），它带来几个显著的问题。

1）从 Kubernetes 的角度来看

- 需要在 Kubernetes 中给各个 SP.赋权，以便它们能够提交代码到仓库。
- Volume Driver 由各个 SP.提供，Kubernetes 的开发者并不了解每个细节，导致这些代码难于维护和测试。
- Kubernetes 的发布节奏和 SP. Volume Driver 的节奏并不一致，随着支持的 SP.增多，沟通、维护、测试的成本会越来越高。
- 这些 SP. Volume Driver 并不是 Kubernetes 本身需要的。

2）从 SP.的角度来看。提交一个新特性或者修复 Bug，都需要将代码提交到 Kubernetes 仓库，在本地编译 Kubernetes 的都知道，这个过程是很痛苦的，这对 SP.而言是完全不必要的成本。

一个统一的、大家认可的容器存储接口越来越有必要。

10.4.2 Container Storage Interface

基于这些问题和挑战，CO.厂商提出 CSI 用来定义容器存储标准，它独立于 Kubernetes Storage SIG，由 Kubernetes、Mesos、Cloud Foundry 三家一起推动。它有两个核心目标：提供统一的 CO.和 SP.都遵循的容器存储接口；一旦 SP.基于 CSI 实现了自身的 Volume Driver，即可在所有支持 CSI 的 CO.中平滑地迁移。

还有一个附带的好处是，一旦 CO.和 SP.都遵循 CSI，就便于将 Volume Driver 解耦到 Kubernetes 的外部（又叫 out-of-tree）。Kubernetes 只用维护 CSI 接口，不用再维护 SP.的具体实现，维护成本大大降低。

CSI 优化下的架构如图 10-15 所示。

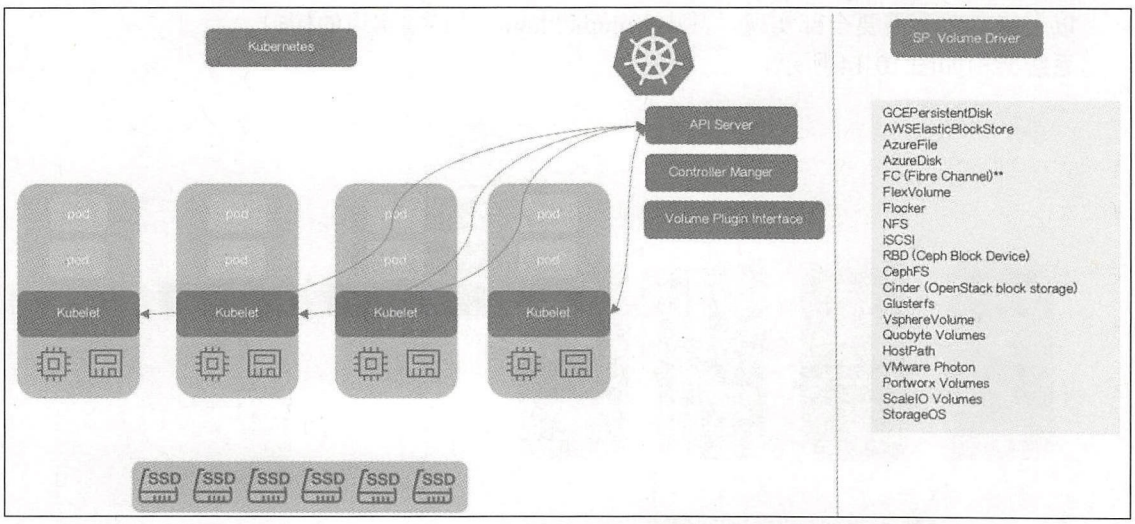


图 10-15 CSI 优化下的架构

可以看到，Kubernetes 和 SP.从此泾渭分明。但事情并没有那么简单，和性能一样，扩展性和解耦也不是凭空出现的。可进一步细化成图 10-16。

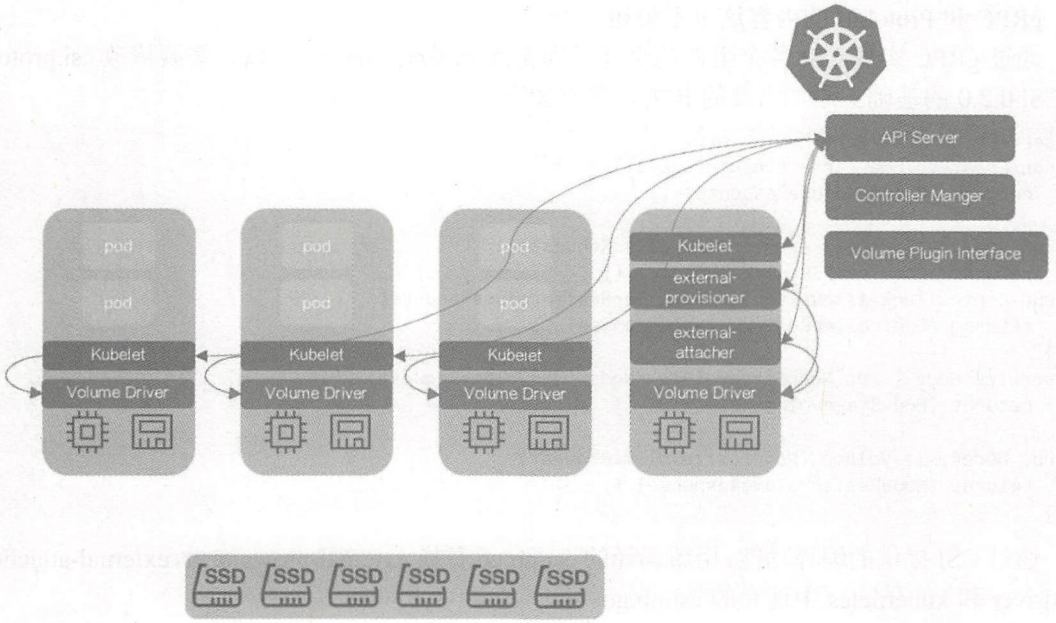


图 10-16　CSI 优化下的架构

1）明显的变化

- Controller Plane、Kubelet 不再直接与 Volume Driver 交互，引入 external-provisioner 和 external-attacher 完成该工作。
- SP. Volume Driver 会由独立的容器运行。
- 为了实现 external-provisioner，external-attacher 和 SP. Volume Driver 的交互引入 gRPC 协议。

2）其他的变化

- 在 Kubernetes 端引入新的对象：CSIPersistentVolumeSource 类型 PV 由 CSI Driver 提供，VolumeAttachment 同步 Attach 和 Dettach 信息。
- 引入新的名称：mount/umount：NodePublishVolume/NodeUnpublishVolume；attach/dettach：ControllerPublishVolume/ControllerUnpublishVolume。

可见，为了达到这个目标，相比原来复杂不少，但收益巨大，Kubernetes 和 SP.的开发者可以专注于自身的业务。

即便如此，在现有的 CSI 上做扩展有时也在所难免。

10.4.3　基于 CSI 和分布式文件系统在 MySQL 上实现 Dynamically Expand Volume

Kubernetes 1.10.2 使用 CSI 0.2.0，其并不包含 Expand Volume 接口，这意味着即便底层的存储支持扩容，Kubernetes 也无法使用该功能，所以需要做点硬编码实现该功能。

1. 扩展 CSI Spec

前面提到在 CSI 中引入了 gRPC，个人理解在 CSI 的场景有如下三个优点：

- 基于 Protobuf 定义强类型结构，便于阅读和理解。
- 通过 stub 实现远程调用，编程逻辑更清晰。
- 支持双工和流式，提供双向交互和实时交互。

gRPC 和 Protobuf 的内容这里不赘述。

通过 gRPC 实现 CSI 各个组件的交互。为支持 expand volume 接口，需要修改 csi.proto，在 CSI 0.2.0 的基础上添加需要的 RPC，重点如下。

```
service Controller {
rpc CreateVolume (CreateVolumeRequest)
  returns (CreateVolumeResponse) {}
……
rpc RequiresFSResize (RequiresFSResizeRequest)
  returns (RequiresFSResizeResponse) {}
rpc ControllerResizeVolume (ControllerResizeVolumeRequest)
  returns (ControllerResizeVolumeResponse) {}
}
service Node { rpc NodeStageVolume (NodeStageVolumeRequest)
  returns (NodeStageVolumeResponse) {}
……
rpc NodeResizeVolume (NodeResizeVolumeRequest)
  returns (NodeResizeVolumeResponse) {}
}
```

通过 CSI 提供的编译功能，用新编译的 csi.pb.go 替换 external-provisioner，external-attacher、csi-driver 和 kubernetes 中现有的 csi.pb.go。

2. 扩展 CSI Plugin

在现有 CSI Plugin 基础上实现接口 ExpandableVolumePlugin。

```
type ExpandableVolumePlugin interface {
VolumePlugin
ExpandVolumeDevice(spec *Spec,  newSize resource.Quantity,  oldSize resource.Quantity)
(resource.Quantity,  error)
RequiresFSResize() bool
}
```

3. 仿照 csiMountMgr 的方式调用 CSI Driver 中的 ControllerResizeVolume。基于 CSI Spec 实现 Storage Driver

CSI Driver 实现如下所有接口：CreateVolume、DeleteVolume、ControllerPublishVolume、ControllerUnpublishVolume、ValidateVolumeCapabilities、ListVolumes、GetCapacity、ControllerGetCapabilities、RequiresFSResize、ControllerResizeVolume。这里涉及比较复杂的调试和适配工作，还有一些其他的工作：

- 定义 CSI 对应的 StorageClass，并设置 allowVolumeExpansion 为 true。
- 启用 Feature Gates：ExpandPersistentVolumes。
- 新增 Admission Control：PersistentVolumeClaimResize。

4. 演示

如图 10-17 所示为扩展效果演示图。

通过扩展 Container Storage Interface 0.2.0，在 Kubernetes 1.10.2 上实现了在线扩容和文件系统扩容。对 MySQL 实例制造两种类型的工作负载：通过键值更新和查询，批量数据加载数据。

通过图 10-18 可以观察到：

- 读数一：MySQL QPS 在正常波动范围内。
- 读数二：持续批量加载数据，MySQL 文件系统容量不断变大。
- 读数三：在 20 分钟内，在线动态扩容两次 Volume 和 Filesystem，过程高效平滑。

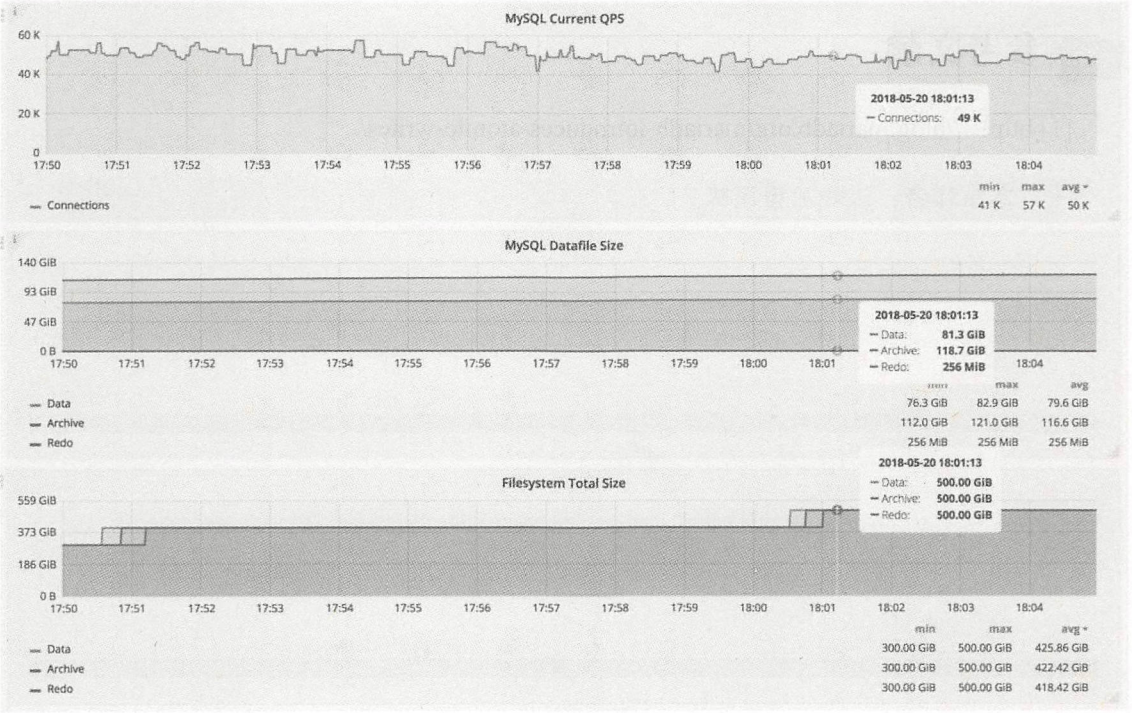


图 10-17　扩展效果演示图

工作到这里基本结束，要改动的地方不少，客观上并不简单。如果没有 Kubernetes 和 CSI，难度会更大。通过此方法可以完成对其他 Storage Vendor 的扩展，譬如 FCSan 和 iSCSI。

10.4.4　对 CSI 的展望

目前 CSI 已发展到 0.2.0 版本，0.3.0 版本也发布在即。0.3.0 版本中呼声最高的特性是 Snapshot。借助该功能，可以实现备份和异地容灾。但是为了实现该功能，在 Kubernetes 现有的 Control-Plane 上还要添加新的 Controller，复杂度会进一步提高。

同时，部分 in-tree Volume Driver 已通过 CSI 迁移到外部，考虑到 Kubernetes 整体的发布节奏和 API 的稳定性，节奏不会太快。

除此之外，CSI 可能还有更多工作要做。以一个高可用的场景为例，当一个 Node 发生故障时，CO 触发 Unmount→Dettach→Attach→Mount 的流程，配合 Pod 的漂移。借助 CSI 定义的接口，Unmount、Dettach、Attach、Mount 由 SP.自身现实。但是 Unmount->Dettach->Attach->Mount 的流程还是有 CO 控制，这个流程并不标准，但是对 workload 又至关重要，如图 10-18 所示。

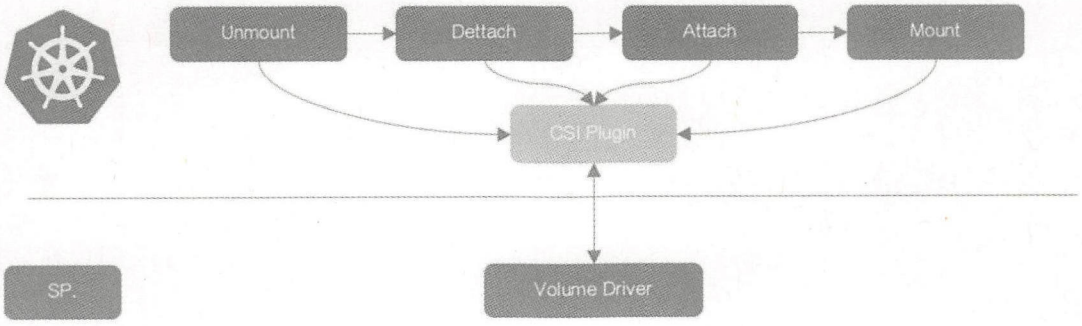


图 10-18　Unmount→Dettach→Attach→Mount 流程

参考文献

[1] http：//blog.mariadb.org/mariadb-introduces-atomic-writes/.

本章作者：沃趣原型团队。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

互联网企业容器技术实践

本书通过容器技术领域的实践者分享各自的实践案例，介绍常见的业务痛点、实现方式、方案选型、遇到的问题和解决方案等。本书主要分为两部分：第一部分是原理篇，简单介绍Docker和Kubernetes的基础知识及原理，包括Docker和Kubernetes是什么、可以做什么及如何使用等；第二部分是案例篇，通过多个实战案例，针对不同的使用场景和业务需求，介绍如何应用容器技术及实现相关需求。本书案例均来自一线真实案例，并有技术人员对技术改造过程的体会和领悟，有较强的借鉴意义和参考价值。

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：陈晓猛 宋亚东
责任编辑：宋亚东
封面设计：李 玲

上架建议：计算机/容器

ISBN 978-7-121-35004-7
9 787121 350047 >
定价：69.00元